Introduction Générale

Définitions

a) **Génie** : ensemble de connaissances et de techniques concernant
la conception,
la mise en œuvre
et les applications de procédés, de dispositifs propres à un domaine déterminé.

b) **Logiciel** : ensemble de :
programmes,
procédés,
règles,
et éventuellement de documents
relatifs au fonctionnement d'un ensemble de traitement de l'information.

b1) Caractéristiques du logiciel :
- produit unique (pas de notion de série/lot) ; il est copiable.
- Immatériel, invisible, abstrait.
- Ne s'use pas, ne se dégrade pas.
- Il s'exécute sur un calculateur.
- Sa fiabilité augmente avec le temps, au fur et à mesure de la maintenance (corrections).
- Le logiciel est facile à modifier.
- La taille du logiciel a de moins en moins d'importance.

b2) Différentes catégories de logiciel
Plusieurs catégories existent dont:
1. Sur mesure : pour un client spécifique
2. Générique : vendu sur le marché (outil de traitement de texte, etc.)
3. Embarqués: s'exécute dans du matériel électronique isolé (téléviseur, téléphone mobile), il est difficile à modifier

c) **Génie logiciel (software engineering)** = science de l'ingénierie du logiciel.
Le génie logiciel : l'ensemble des activités
de conception et de mise en oeuvre des produits et des procédures pour rationaliser
la production du logiciel et son suivi (maintenance).

Pourquoi le génie logiciel ?
Crise ➔ naissance du génie logiciel
- peu de fiabilité du logiciel,
- difficulté de livrer des logiciels satisfaisant un cahier de charges dans les délais prévus.

Le génie logiciel s'intéresse :
- aux différentes théories.
- aux méthodologies (techniques, méthodes).
- aux outils.
- A l'organisation générale.
- A la gestion de projets.

Constat

- les logiciels deviennent de plus en plus importants en taille et en complexité.
- Les coûts des logiciels connaissent une très forte croissance dépassant le coût du matériel.
- La fiabilité est plus importante qu'efficacité.
- Les coûts de maintenance dépassent ceux du développement.

Bilan

Etude sur 487 sites de toutes sortes de développement de logiciels
70% du coût du logiciel est consacré à la maintenance.

Coût de logiciel = coût de développement + coût de maintenance.

42% des modifications sont dues à des demandes des utilisateurs.
30% des exigences de spécification changent entre la première édition d'un document et la première sortie du produit.
17% des changements de format des données.
12% problèmes d'urgence résolus à la hâte.
9% débogages de programmes.
6% des changements de matériels.
6% problèmes de documents.
4% amélioration de l'efficacité.
3% autres.

Certaines organisations de production de logiciel consacrent la quasi-totalité de leurs efforts à la maintenance des produits livrés.

Le génie logiciel se propose d'aider à :

- organiser des projets (planification et communication pour le travail en équipe)
- estimer les coûts et les délais.
- gérer les ressources (humaines, matérielles et logicielles)
- suivre l'évolution du logiciel (maintenance).
- documenter, définir et respecter les normes.

d**)** **Norme** : est un document qui apporte des réponses à des questions techniques et commerciales que se posent de façon répétée les acteurs sur
des produits,
des biens d'équipements
ou des services.

Elle est élaborée en consensus par l'ensemble des acteurs d'un marché (producteurs, utilisateurs, consommateurs, pouvoirs publics).

Quelle est l'utilité d'une norme ?

- Echanger : la norme permet à l'entreprise de développer ses marchés (même norme) et de clarifier ses transactions (produit défini selon le besoin de l'utilisateur – rapprochement entre le fournisseur et le consommateur).
- Développer : la norme contribue au développement par la satisfaction du client (maîtrise des caractéristiques techniques des produits).
- Orienter : la norme contribue à l'information de l'utilisateur (aide au choix des produits de garantie).
- Innover : la norme permet à l'entreprise d'innover, de faire évoluer les produits et de faciliter l'insertion de l'innovation sur le marché.

Conception d'un système

1. **Les enjeux de l'approche systémique.**
2. **Définition d'un système.**
3. **Classification des systèmes.**
4. **Principe des systèmes**
5. **Caractéristique d'un système.**
6. **Cycle de vie d'un système.**
7. **Conception d'un système**
8. **La conception statique d'un système**
9. **La conception dynamique d'un système**

1. Les enjeux de l'approche systémique

- Difficulté de la phase de conception (préliminaire et détaillée).
- Double compétence de l'architecte (côté matériel et logiciel).
- Choix du matériel et du logiciel pour construire le nouveau système.

2. Définition d'un système

a) Un système est un ensemble composite de personnels, matériels et de logiciels organisés pour que leur interfonctionnement permette, dans un environnement donné, de remplir les misions pour lesquelles il a été conçu [Meinadier,98].

b) Un système est un ensemble
d'équipements,
de savoir faire,
de techniques
capable de jouer un rôle opérationnel ou d'apporter un soutien logistique ou les deux à la fois [Jaulent, 94]

L'accent sera mis sur les systèmes d'information

- les systèmes d'information font partie de systèmes plus étendus (imbrication).

Exemple : le système de paie fait partie du système de personnel.

- Souvent un nouveau système doit remplacer un autre système existant et on doit connaître son fonctionnement.
- Tous les systèmes possèdent des similarités (ressemblances).

3. Classification des systèmes

- Systèmes naturels inertes (système solaire)
- Systèmes naturels vivants (organismes)
- Systèmes artificiels, conçus par des humains :
  - d'ordre social (lois, douane)
  - transports (réseau routier)
  - fabrication (chaîne de montage)

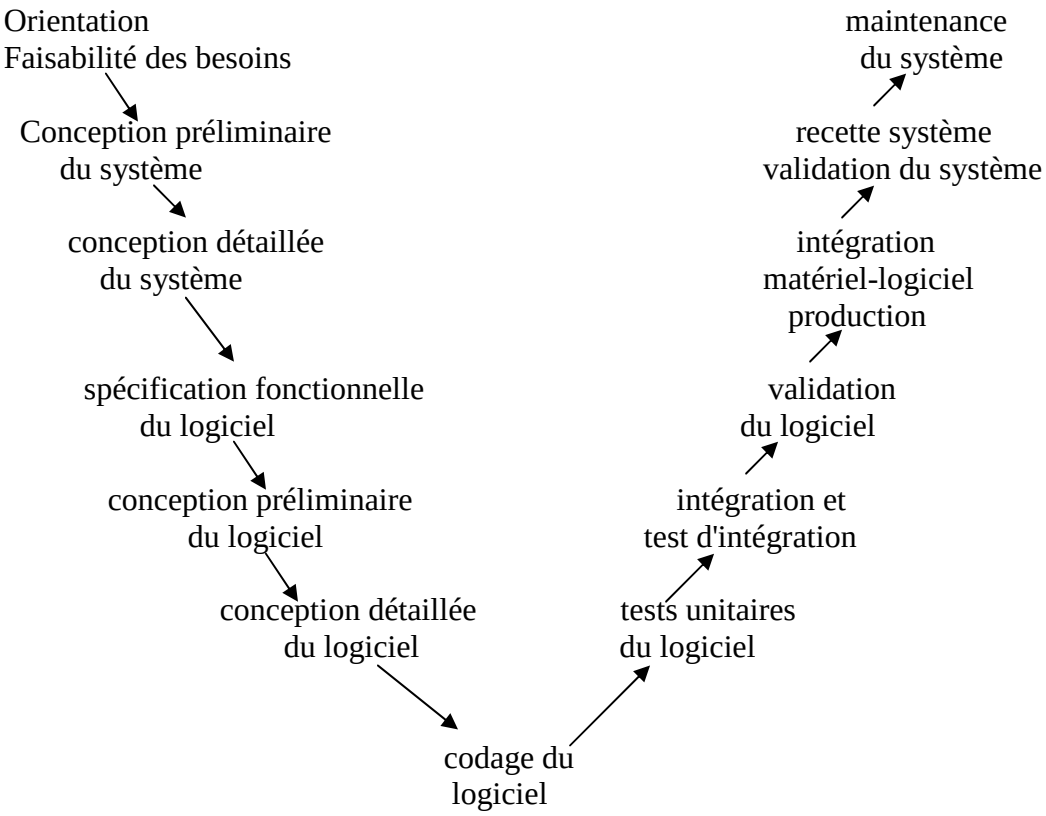
4. Principes des systèmes

- Plus un système est spécifique, moins il est capable de l'adapter.
- Plus un système est large, plus il demande des ressources pour son entretien.
- Un système fait partie d'un système plus large, on peut le diviser en sous-systèmes.
- Un système prend de l'expansion avec l'âge.

5. Caractéristiques d'un système

- Sa structure: les objets qui le composent,
- Son évolution: les états successifs dans lesquels il se trouve,
- Ses fonctions: ce qu'il sait faire.

6. Cycle de vie d'un système



Discussion

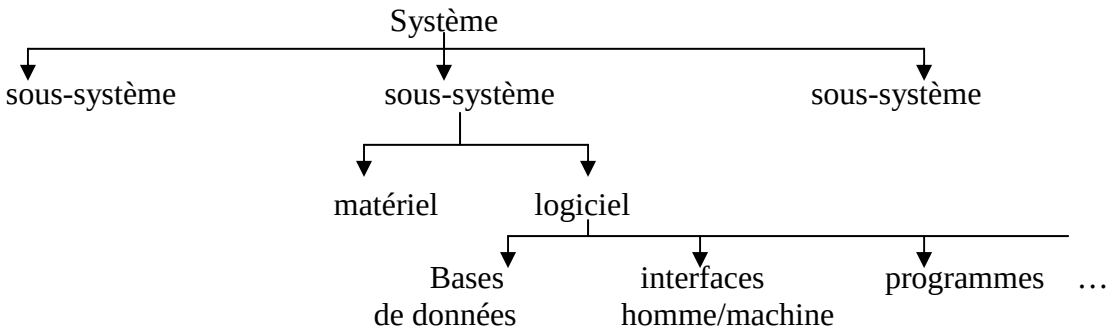
Etude de la faisabilité
Conception préliminaire du système } Spécification
Conception détaillée du système

Spécification fonctionnelle du logiciel
Conception préliminaire du logiciel } Production
Conception détaillée du logiciel
Codage

Tests unitaires du logiciel
Intégration et test d'intégration
Validation du logiciel } Contrôles
Intégration matériel-logiciel
Production

Validation du système } Vente-utilisation
Maintenance du système entretien

Le développement du logiciel s'insère dans le cadre du développement du système.
Il est important de comprendre l'articulation entre les deux cycles.
Un projet de développement de système commence par une étude de la faisabilité qui est suivie par une phase de conception.
Lors de cette phase, on décrit l'architecture du système et on alloue les fonctions du système au matériel, au logiciel et aux hommes.
Cette démarche aboutit aux besoins en logiciel qui constitue l'élément technique essentiel pour démarrer le développement en logiciel.



7. Conception d'un système

- Elle concerne les deux étapes de conception: conception préliminaire et la conception détaillée (du système).
- Les méthodes de conception retenues sont celles imposées par les normes.
- La conception présente deux aspects: statique et dynamique.

Aspect statique : base contractuelle entre le demandeur et le réalisateur.
Préciser ce que doit faire le système (le quoi) et non le comment
STB, SADT, E-A

Aspect dynamique : maquettes ou prototypes, réseaux de Petri, SYS-P-O, SREM

8. La conception statique d'un système

8.1 Spécification technique des besoins (STB)

Ils sont fournis par un document dont le plan type est inspiré de la norme IEEE std830

A. Introduction

Le plan type est établi selon un contrat d'accord entre le demandeur et le réalisateur et permettant de préciser ce que fait le système.
Ce document sert de preuve :

- le projet est économique
- la bonne maîtrise du cycle de développement (par des estimations et la planification)
- les coûts de développement réduits
- le support à l'assurance qualité du logiciel

B Le contexte de réalisation

B.1 objectifs : système à construire ou à améliorer
B.2 hypothèses : à respecter et pouvant remettre en cause tout ou une partie de la réalisation des spécifications.
Exemple : livraison des algorithmes par le client

B.3 bases méthodologiques : précisent
- les procédures
- les méthodes
- les normes
- les langages

à utiliser pour mener à bien le projet.

C Description générale

C.1 Environnement du projet: place du projet par rapport à l'environnement dont il dépend et par rapport aux projets en parallèle.

C.2 Fonctions générales du système: simple description des fonctions utilisateurs du système ainsi que les principaux objets qu'il manipule.

C.3 Caractéristiques des utilisateurs: identifier les différents utilisateurs (s'ils existent) et préciser pour chacun les caractéristiques qui affectent l'interface utilisateur (menu) :
- connaissance ou non de l'informatique
- expérience de l'application
- utilisateurs réguliers / occasionnels

C.4 Configuration du système : rubrique décrivant une configuration générale du système (avec référence à un document spécifique).

C.5 Contraintes de développement, d'exploitation et de maintenance

C.5.1 Contraintes de développement sont liées au(x):
- matériel
- langages de programmation imposés
- logiciel de base
- algorithmes imposés
- délai de réalisation

C.5.2 Contraintes d'exploitation sont liées au(x):
- règles de gestion du système
- affectation des responsabilités des utilisateurs
- planning d'exploitation (jour, nuit,)
- problème de sauvegarde et reprise
- problèmes de sécurité et d'intégrité

C.5.3 Contraintes de maintenance sont liées aux procédures de maintenance
- curative
- adaptative
- évolutive
- perfective

D Description des interfaces externes du logiciel

D.1 interface matériel / logiciel : décrit le matériel informatique (périphérique) et les procédures d'échange avec le matériel non informatique (capteurs, caméra) ou les systèmes environnants.

D.2 interface homme / machine :
- renseignement sur l'ergonomie du système:caractéristiques des messages d'erreur (clarté), commande du logiciel par menu, par " icônes " avec souris.
- mode d'apprentissage éventuel
- le niveau d'intelligence des interfaces H/M.

D.3 interface logiciel / logiciel concerne:
- les moyens d'accès à des systèmes SGBD
- les procédures d'échange de messages

E. Description des objets : identifie les objets qui composent le système.
E.1 définition des objets
- objet-I: nom de l'objet, attributs, valeur de chaque attribut, classe d'objets.
- contraintes sur l'objet-I: contraintes sur les attributs de l'objet.

F. Description des fonctions: identifie et décrit chaque fonction à chaque objet
F.1 Définition des fonctions
- identification-I: nom de la fonction, son rôle, sa présentation générale.
- description-I: traitement associé à la fonction et à ses interfaces.
- contraintes opérationnelles sur chaque fonction-I.

F.2 Conditions particulières de fonctionnement: dispositions prises dans les conditions particulières de fonctionnement.
F.2.1 performances: spécifications temps réel liées à l'utilisation du système (termes mesurables)
- du point de vue utilisateur: temps de réponse souhaité, fréquence d'utilisation.
- du point de vue de l'environnement: fréquence moyenne d'acquisition d'états ou de mesures.

F.2.2 capacités : limites des problèmes traités par le logiciel et les limites des extensions possibles.
F.2.3 mode de fonctionnement : mode d'exploitation du système
- la mise sous tension
- l'arrêt
- la reprise de secours

F.2.4 contrôlabilité: possibilité de suivre l'exécution d'un traitement
Exemple
Pour une base de données, connaître à la suite d'une modification, les données avant et après la modification
F.2.5 sécurité : niveau de confidentialité du système (utilisation des mots de passe)
F.2.6 intégrité: protection contre la perte d'information
F.2.7 conformité aux standards: références aux standards ou aux normes
F.2.8 facteurs de qualité : références à un plan de qualité du logiciel

G. Justification des choix effectués: elles concernent les spécifications envisagées et délaissées, les alternatives possibles pour conserver une trace de l'évolution de la pensée.

H. Glossaire: comprend la définition
- des termes courants utilisés hors de leur sens commun
- des termes techniques
- des abréviations
- des sigles

employés dans les documents.

I. Références
- références documentaires sur les méthodes
- références bibliographiques
- sources d'obtention des documents

I.1 annexes
Une annexe comprend le modèle complet de spécification système sélectionné (modèle SADT).
I.2. index: indique les pages où sont mentionnés les sujets et les termes les plus importants du document

8.2 L'analyse structurée et modélisation technique (S.A.D.T)

8.2.1. Historique

- développée par Douglas T.Ross et testée entre les années 1969 et 1973 (implication dans un projet aérospatial).
- introduite dans le marché en 1975.
- utilisée, en 1981, par plus de 50 sociétés, sur 200 projets impliquant plus de 2000personnes et couvrant plusieurs domaines (téléphonies, informatique, aérospatial)
- diffusée actuellement sous le nom de IDEF-0 (Icam DEFinition) numéro 0.

Le programme ICAM (Integrated Computer Aided Manufacturing) de U.S. Air Force retient une partie de SADT pour da méthode IDEF-0, principalement les actigrammes où les boites représentent les actions et les flèches les flots de données. La partie datagramme ne fait pas partie de IDEF-0

8.2.2. Description de la méthode

- SADT: Structured Analysis and Design Technique en américain, il faut comprendre analyse structurée et modélisation technique.
- SADT est une méthode d'analyse structurée. C'est une méthode qui cherche à établir la communication entre les demandeurs et les utilisateurs d'une part et les concepteurs et les réalisateurs d'autre part.
- SADT est une méthode de spécification fonctionnelle du système (permettant de préciser les fonctions que le système doit assurer).
- SADT conduit à la représentation d'un ou de plusieurs modèles.
- Un modèle SADT est une description complète, concise et consistante d'un système qui est développée dans un objectif précis.

SADT =
un formalisme pour la définition des fonctions d'un système (SA)➔ modèles
une démarche, un mode d'emploi pour utiliser au mieux ce formalisme (DT) ➔processus d'élaboration des modèles.

8.2.3 Domaines d'application

Les principaux domaines actuels d'application de la méthode SADT sont:

- systèmes d'information
- systèmes de production
- systèmes aéronautiques, militaires, etc.

8.2.4 Objectifs de SADT

- analyser et comprendre les systèmes complexes (peut aussi concerner les petits systèmes).
- fournir les résultats de l'analyse et de la conception dans une notation claire et précise.
- assurer la qualité

- répartir le travail d'équipe et coordonner les efforts.
- gérer le projet

8.2.5 Les concepts de la méthode SADT

8.2.5.1. Modéliser pour comprendre

SADT permet d'analyser un système en construisant un modèle pour

- rendre possible la compréhension profonde du problème
- permettre à tous les acteurs du projet de partager un langage commun

SADT ne s'utilise pas pour décrire la structure d'un système, quels sont ses composants mais pour comprendre ce que fait le système, les besoins auxquels il doit répondre, quelles opérations doivent être appliquées aux entrées pour produire des sorties.

8.2.5.2. Discipliner la démarche d'analyse

L'analyse de tout système est faite de manière structurée:

- descendante
- modulaire
- hiérarchique

L'utilisation de SADT part d'une vision globale du système
➔ Le système global correspond à un module.
Elle se poursuit selon une approche descendante. Chaque module est éclaté en sous-modules: chaque boite est un ensemble de boites (hiérarchisation).
Le modèle est structuré de façon à fournir progressivement un nombre croissant de détails. Pour chaque nouveau niveau, 3 à 6 nouvelles boites sont introduites.
Cette décomposition d'un problème en modules, boites, flèches constitue ce que l'on nomme un diagramme.

8.2.5.3. Séparer le quoi du comment

Spécifier un système consiste à fournir une description fonctionnelle du système (quoi) indépendamment des solutions envisageables pour sa réalisation (comment).
L'analyse fonctionnelle ne donne aucune orientation sur les choix effectués ultérieurement en matière de méthode de conception, de langage de programmation, etc.

8.2.5.4. Modéliser la réalité

La représentation du monde qui nous entoure est constituée d'objets (données) et d'actions (activités). SADT aborde la modélisation du système en utilisant les actigramme pour les activités et les datagrammes pour les données.

8.2.5.5 Formaliser de manière graphique

Les langages naturels ne permettent pas une représentation claire et précise (non ambiguë) des résultats d'analyse. Les modèles SADT sont basés sur une représentation graphique (boites / flèches) ainsi que des mots destinés à :

- fournir les détails de manière progressive
- encourager la précision
- mettre l'accent sur les interfaces des modèles
- fournir un vocabulaire puissant d'analyse.

Le symbolisme SADT a pour caractéristique:

- un ensemble limité de primitives de base à partir desquelles l'auteur peut construire méthodiquement des structures de toutes tailles.
- une lecture de droite à gauche et de haut en bas.

8.2.5.6. Travailler en équipe

L'analyse d'un système complexe nécessite une équipe de travail discipliné et bien coordonné. SADT crée l'environnement nécessaire à cela en

- offrant un langage commun
- précisant le rôle de chacun dans l'équipe
- constituant la documentation contenant tous les résultats (définitifs et intermédiaires).
- insistant sur le cycle auteur-lecteur

L'équipe SADT comprend :

- auteur ou analyste : personnes étudiant les besoins et analysant les fonctions du système.
- commentateurs ou lecteurs : personnes critiquant le travail des autres auteurs.
- experts ou spécialistes : personnes fournissant les informations sur le système.
- comité technique : équipe critiquant l'analyse à chaque étape de la décomposition
- bibliothécaire : responsable de la tenue à jour de l'ensemble des documents du projet et de leur circulation.
- chef de projet : il a la responsabilité technique du projet.

8.2.5.7. Consigner par écrit

SADT oblige à consigner par écrit tous les choix réalisés pendant l'analyse. Les documents ainsi crées sont accessibles à tous pour être revus et éventuellement modifiés.
Cette approche se base sur le cycle auteur-lecteur

- rédaction d'un document (auteur)
- lecture du document (lecteur)
- rédaction des commentaires et critiques destinés à l'auteur (lecteur)
- évolution et rédaction d'une nouvelle version (auteur)

Toutes les versions intermédiaires sont conservées (de manière automatique ou manuelle)

8.2.6. Syntaxe de SADT

Un modèle SADT est composé de

- actigrammes ou diagrammes d'activités
- datagrammes ou diagrammes de données
- textes explicatifs sur les actigrammes
- diagrammes Pour Explication Seulement appelés PES
- liste hiérarchique : schéma de la hiérarchie du système analysé
- glossaire
- conditions d'activation

6.2.6.1 L'actigramme

Dans un actigramme, chaque activité est identifiée par un verbe.

a) L'activité :

- crée, génère une donnée en sortie
- transforme, modifie, change l'état d'une donnée d'entrée en donnée de sortie.

b) Les données d'entrée : données qui sont transformées, modifiées ou sollicitées par l'activité.

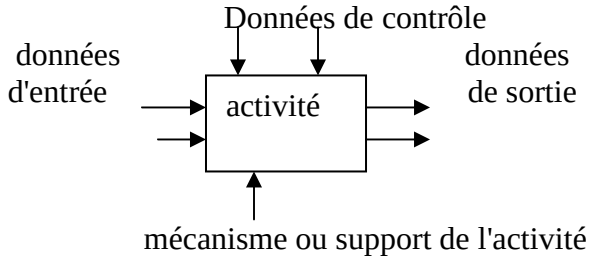
c) Les données de contrôle : ce sont des
- données qui contrôlent comment la transformation doit s'effectuer,
- données qui ne sont pas modifiées,
- données qui présentent une contrainte sur la façon dont l'activité sollicite l'entrée.

Les données de contrôle permettent de déclencher l'activité ou peuvent simplement être utilisées comme paramètres contrôlant la transformation de la donnée d'entrée en donnée de sortie.

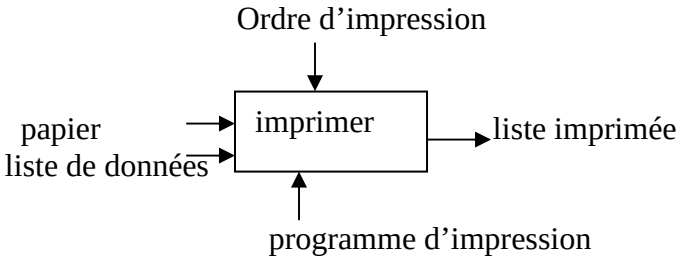
d) Les données de sortie: données créées ou générées par l'activité

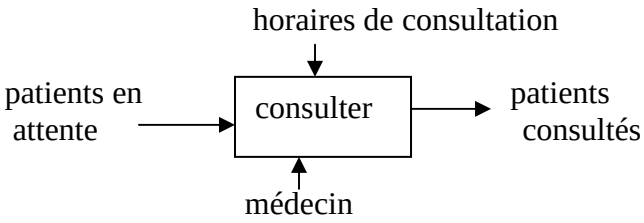
e) Les mécanismes : indiquent quelles ressources (humaines ou matérielles) seront utilisées pour effectuer l'activité.

C'est <u>le comment</u> est réalisé l'activité ou <u>le qui</u> réalise l'activité.



Exemples

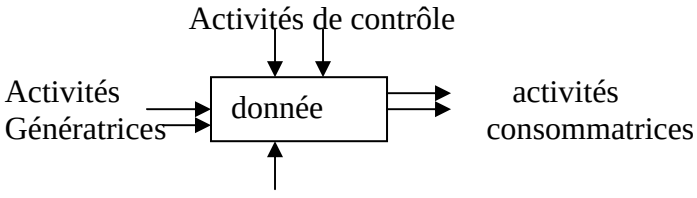




<u>8.2.6.2. Le datagramme</u>

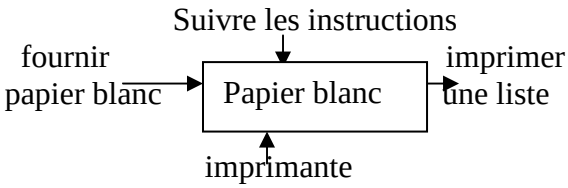
Dans un datagramme, chaque donnée est identifiée par un nom.

Les activités en entrée sont celles qui créent ou génèrent la donnée, les activités en sortie sont celles qui l'utilisent ou la consomment sous le contrôle des activités de contrôle.



Support des données (dispositif de mémorisation de la donnée)

Exemple



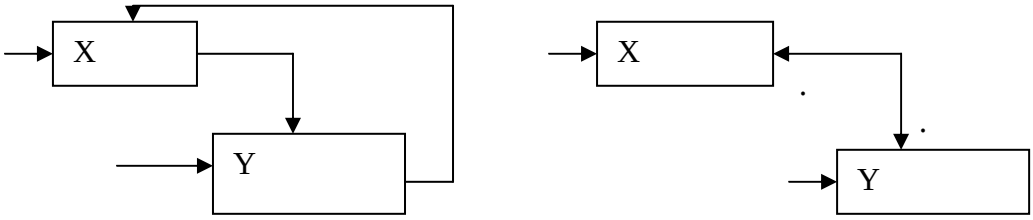
Remarques

- Dualité activité / donnée

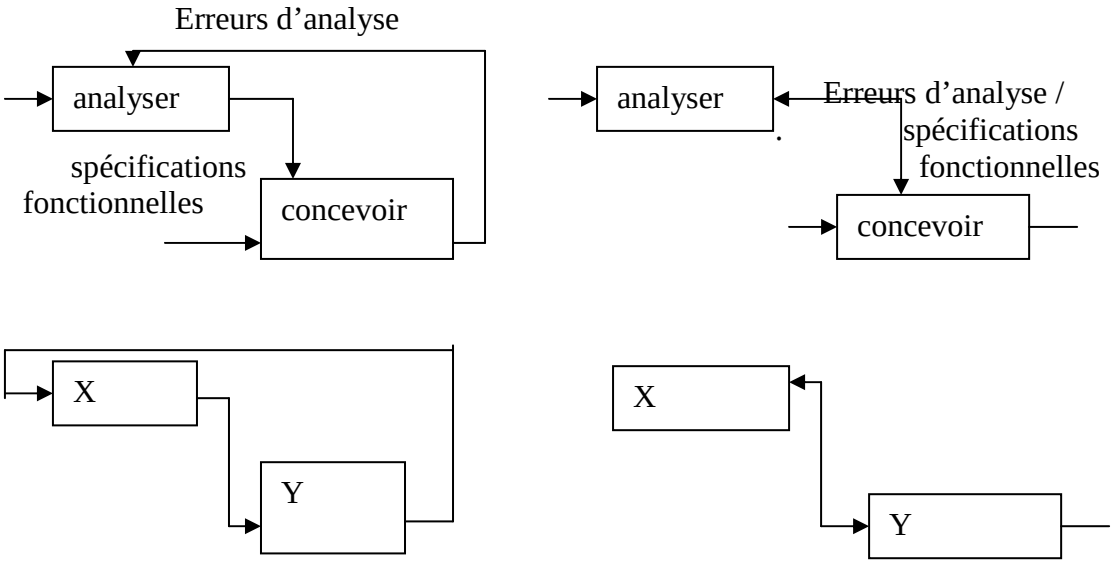
Activité : ce que fait le système, les opérations ou les traitements qu'il réalise.
Donnée : ce que le système manipule, utilise ou produit.
Les activités et les données sont indispensables et prennent leur sens l'un par l'autre.
En pratique, on n'utilise pas les datagrammes SADT mais le formalisme E-A.

- Règles de construction des diagrammes

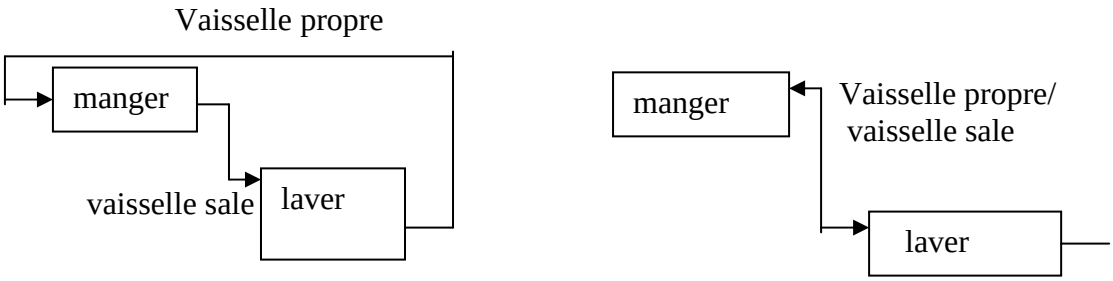
Chaque flèche entrant ou sortant de sa boite-mère doit se trouver sur le diagramme enfant.
Les supports peuvent ne pas être mentionnées si cela n'éclaire pas la compréhension.
On ne mentionne que les éléments nécessaires (ce que l'on veut montrer).
Lorsque la relation est à double sens (entrée réciproque ou contrôle réciproque), on utilise une double flèche avec un point à droite ou sur la pointe des flèches concernées.



Exemple



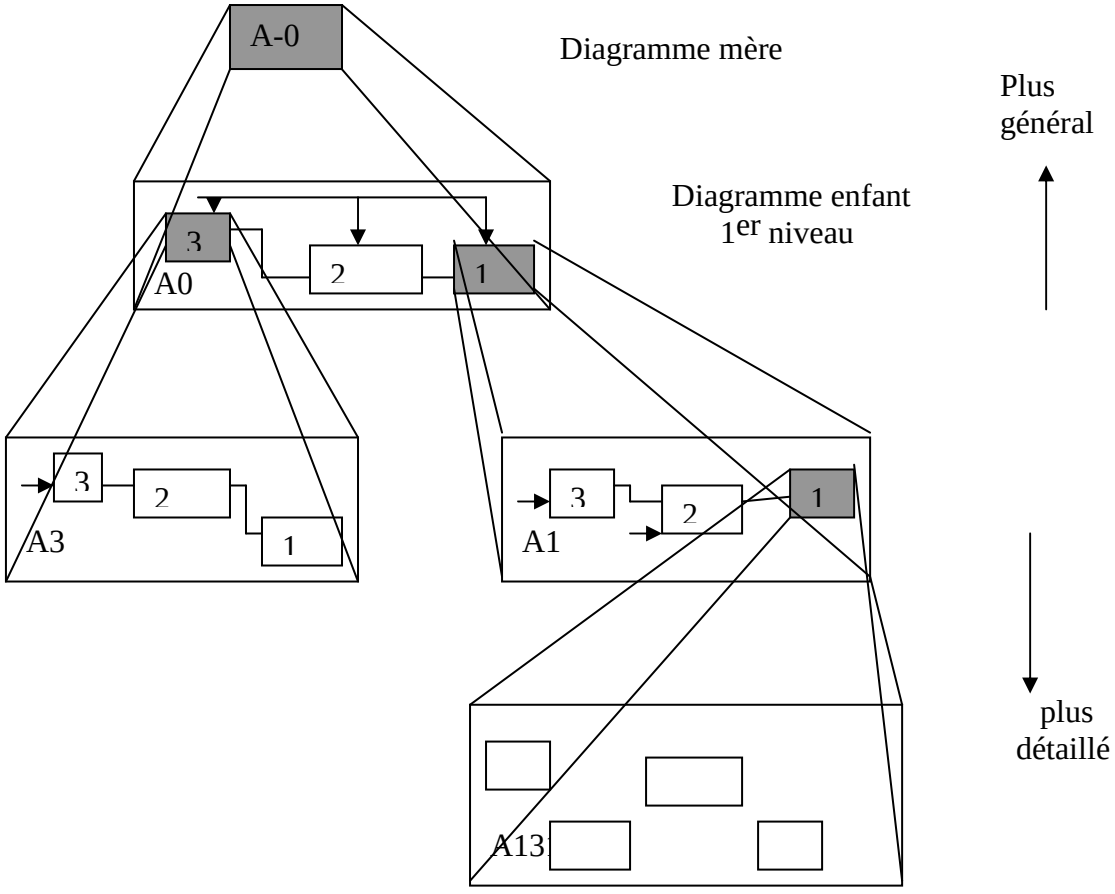
Exemple



8.2.6.3. La liste hiérarchique

Les diagrammes d'un modèle SADT sont numérotés de façon précise. Le premier niveau représente le système global, il porte le numéro A-0 (moins 0) pour les actigrammes et D-0 (moins 0) pour les datagrammes. Il sera décomposé au niveau A0 (resp D0).
Cette boite comprend les nœuds portant les n° A1, A2, ….An (resp. D1, D2, …Dn) qui à leur tour seront décomposés et ainsi de suite.

L'organisation hiérarchique des diagrammes :



8.2.6.4. Les textes explicatifs sur les diagrammes : qui accompagnent les diagrammes pour représenter brièvement des généralités sur le diagramme et les faits auxquels l'auteur accorde un intérêt.

8.2.6.4. Les diagrammes Pour Explication Seulement PES : Ils permettent de clarifier un aspect particulier du système.

8.2.6.5. Le glossaire :
- regroupe les définitions des activités ou des données qui apparaissent sur chaque diagramme.
- Explique le sens des termes techniques, des termes généraux utilisés de manière spécifique, des abréviations.

L'existence d'un glossaire autorise l'emploi des labels (étiquettes) brefs pour les boites et les flèches puisqu'il fournit des explications exactes de leurs caractéristiques (domaine des valeurs).

8.2.6.6. Les conditions d'activation : dans les actigrammes, elles permettent de spécifier dans quelles conditions une boite sera activée et ce qu'elle produit.

8.2.7. Conclusion

L'analyse fonctionnelle selon les concepts et les modèles de SADT permet :
- d'étudier et de préciser les fonctions de chaque élément constituant le système.
- D'aider le concepteur à faire une numérotation détaillée des différents constituants.
- D'orienter l'architecture vers un ensemble de configurations modulaires et hiérarchiques.
- D'aboutir à une analyse efficace dans un environnement stable.

Cependant,
- Chaque changement ou évolution du système contribue à rendre les réseaux de communication entre les fonctions difficile à gérer.
- Le comportement temps réel du système dès la conception n'est pas défini, il faut le compléter par un modèle qui le permet.
- La validation du monde s'effectue par différentes personnes ce qui donne une validation limitée.

8.3 Le modèle entité-association

8.3.1. Introduction

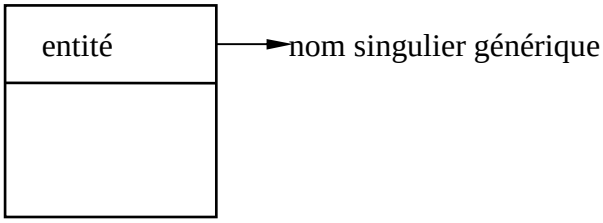
Le formalisme entité-association (E-A en français et E-R en anglais pour Entity-Relationship) est l'un des formalismes retenus par ISO (International Standard Organisation) pour décrire l'aspect conceptuel des données lors de la conception préliminaire et détaillée d'un système d'information. Il est utilisé par plusieurs outils d'aide à la conception des bases de données (merise par exemple). De nombreuses variantes ont été proposées dont celle de Peter Chen.
Le modèle E-A propose une description sur la base de trois (3) concepts : l'entité, l'association et l'attribut (ou propriété).

8.3.2. Concepts de base de la méthode E-A

8.3.2.1 Les entités

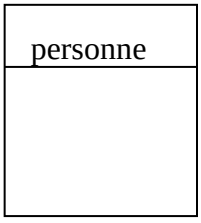
Une entité est le regroupement par la pensée d'un ensemble d'objets ayant des caractéristiques communes.
Le caractère fondamental d'une entité est que ses exemplaires existent en eux mêmes sans référence à d'autres objets.

Représentation d'une entité



Entité ⟶ un type / une classe / un ensemble

Exemple



Ali est un exemplaire de type personne

Comment trouver les entités (dans un énoncé) ?
Une chose est une entité donc on peut donner trois (3) exemples de cette chose.
Si chacun de ces exemples est un objet autonome, la chose est une entité.
Ali, Mohamed, Monsieur sont trois exemples de personnes ➔ personne est une entité.

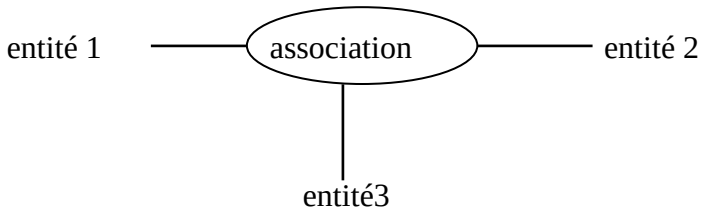
Une entité est pourvue d'une existence propre.

8.3.2.2 Les associations

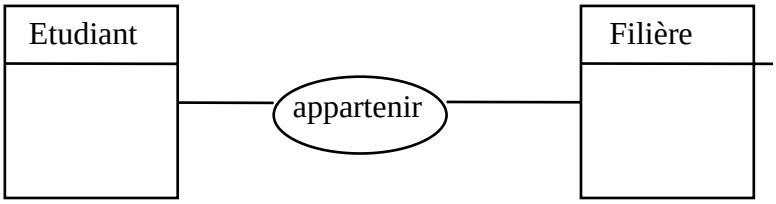
Les entités sont reliées à l'aide de pattes à l'association. Le nombre de pattes d'une association est égal à sa dimension.
Une association binaire donc deux (2) pattes.
Une association ternaire donc trois (3) pattes, etc.

En pratique, les associations de dimension 2 sont majoritaires, parfois de dimension 3 et très rarement de dimension 4 (elles doivent être obligatoirement décomposées).

Représentation d'une association



Exemple :



Comment trouver les associations (dans un énoncé) ?
Une chose est une association, donc on peut donner trois (3) exemples.
Pour chacun de ces exemples, on fait référence à au moins deux autres choses
(impossible d'imaginer l'existence de ces exemples sans ces autres choses), donc cette chose est une association
Exemples :
Ali appartient à la filière « informatique », Mohamed appartient à la filière « physique », etc,
➔donc appartenir est une association.

Une association est dépourvue d'une existence propre.

Cardinalité d'une association pour une entité
- la cardinalité maximale (minimale) d'une association donnée est le nombre maximum (minimum) que l'on peut trouver pour un exemplaire d'entité.

8.3.2.3 Les attributs

Un attribut permet d'affecter une propriété individuelle à un exemplaire d'entité ou d'association.
Un attribut a un nom et prend une valeur pour chaque exemplaire d'entité ou d'association.

Exemple
l'attribut " âge " défini par l'entité "personne " affecte à Ali la valeur « 20 ans »

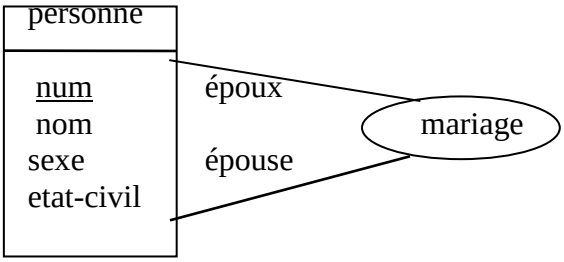
Comment trouver un attribut (dans un énoncé) ?
Une chose est un attribut si on trouve des exemples et pour chacun on fait référence à une autre chose.

Exemples:
L'âge de Mohamed, l'adresse de Mohamed, etc. ➔ donc âge, adresse sont des attributs pour l'entité " personne ".

L'identifiant d'une entité est un attribut particulier qui à chaque valeur de l'attribut correspond un et un seul exemplaire de l'entité.

Exemples
L'entité " personne " a pour identifiant l'attribut " numéro-de-sécurité-sociale ".
L'entité " voiture " a pour identifiant l'attribut " numéro-immatriculation ".

8.3.3. Contraintes d'intégrité
Les concepts d'entité, d'association, d'attributs sont insuffisants pour décrire ce qui caractérise les données d'un schéma E-A.



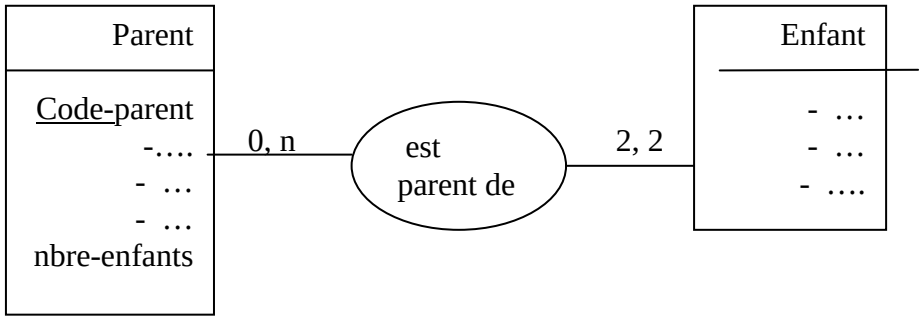
Si une occurrence de Personne participe à l'association « mariage » ➔ la valeur de son attribut « etat-civil » ← « marié ».

Un formalisme possible (inspiré de la logique du 1er ordre) pour cette règle est :
∀x, y ∈ Personne
< x, y > ∈ mariage ➔ x.etat-civil← " marié " et
y.etat-civil ← " marié"
Une telle règle est appelée contrainte d'intégrité.
Si les valeurs de la base de données ne satisfont pas ces contraintes, alors il y a une erreur dans la base de données.
On dit que la base de données est incohérente.

8.3.3.1 Contraintes d'intégrité sur les attributs.
Elles permettent de limiter les valeurs possibles d'un attribut à certaines valeurs du domaine.
Exemple
L'attribut « âge » est compris entre [0..130].

8.3.3 2 Contraintes d'intégrité sur les cardinalités
Elles permettent de limiter les cardinalités des entités, des associations.

Exemple



L'attribut "nbre-enfants" représente le nombre de participation de l'entité « parent » à l'association "est parent de"

8.3.4. Conclusion
Le formalisme E-A peut être utilisé lors des phases de conception statique d'un système pour identifier les objets du monde réel et établir leur relation.
Le modèle conceptuel est plus stable. Le fait de passer d'un type de SGBD à un autre ne remet pas en cause le schéma E-A.

9. La conception dynamique d'un système

9.1 Le prototypage d'exigence

9.1.1 Introduction

Afin de mieux cerner le problème et sa solution et dans l'optique de limiter le nombre d'erreurs dans le logiciel final, une technique utilisée est le prototypage permettant la construction d'implémentations partielles de systèmes.
S'il s'agit d'une implémentation complète, on parle de représentation du système tout entier et non de sa maquette.
Deux types de prototypes sont à distinguer :

- Les maquettes jetables : la construction du prototype sert à approfondir la compréhension du problème et à trouver sa solution puis on les jette après usage.
- Les prototypes évolutifs : dans ce cas, le prototype n'est pas abandonné mais plutôt adapter à la nouvelle configuration des besoins.

9.1.2 Qu'est ce qu'un prototype ?

Un prototype correspond à un besoin de vérifier le comportement réel du produit en cours de développement. Il sert à la communication entre le spécifieur et le réalisateur.
C'est une approximation de l'interprétation que le réalisateur fait de la spécification.
Le prototypage consiste en quatre étapes :

- la sélection fonctionnelle (choix des fonctions à réaliser)
- la construction
- l'évaluation (feedback sur le développement)
- l'utilisation ultérieure du prototype :
  soit est mis de côté (maquette jetable)
  soit fait partie intégrante du produit (prototype évolutif)

9.1.3 Impacts du prototype sur les exigences et sur le cycle de vie

a) concernant les exigences :

Pour les systèmes complexes, les exigences :

- sont généralement comprises de manière superficielle
- évoluent ou changent tout le long du cycle de développement
- risquent d'être remises en question lors de l'utilisation du système
  ➔ d'où l'intérêt du prototypage (prototype rapide).

b) concernant le cycle de vie

Une maquette jetable peut être construite pendant n'importe quelle phase de la branche descendante du cycle de vie du système / logiciel.

b1. Pendant la phase de spécification

La construction d'une maquette provisoire aide l'utilisateur à :

- fixer la faisabilité des exigences
- valider la nécessité de toutes les fonctions
- découvrir les exigences manquantes
- déterminer la convivialité de l'interface utilisateur

A l'aide de cette expérience, l'équipe de réalisation peut compléter les documents de spécification du système / logiciel et parvient à un produit correctement spécifié.

b2. Pendant la phase de conception préliminaire
La construction d'une maquette provisoire permet de valider les propriétés d'une architecture particulière.

b3. Pendant la phase de conception détaillée
Elle permet de vérifier la suffisance des algorithmes.

Remarque :
Une maquette doit être rapide et provisoire.
Rapide ➔ résultats immédiats.
Provisoire ➔ pas de justification sur la qualité d'un produit jetable,
pas de conception,
pas de plan de test,
codage dans un langage le plus simple possible

Comment élaborer une maquette pour une spécification ? Voici un scénario possible :

- écrire les documents C-CH-F1 et DSL1
- implémenter une maquette conforme aux exigences,
- présenter la maquette de la future application à un utilisateur potentiel,
- réécrire les documents C-CH-F2 et DSL2 (en fonction des remarques des utilisateurs)
- développer le produit.

Pour les prototypes évolutifs, leur construction n'est ni provisoire ni rapide.
Ils évoluent vers un produit final d'où la nécessité d'intégrer les facteurs de qualité du produit final. Pour assurer

- la qualité,
- la maintenabilité
- et la faisabilité d'un produit logiciel, il faut respecter le cycle de vie en V.

D'où création de tout document nécessaire ainsi que la présence obligatoire des revues de qualité.

9.1.4 Impacts sur la productivité et le succès du produit

Qu'est ce que la productivité ?
La productivité c'est la quantité de travail fournie par une personne pendant une période de temps donnée.
La productivité se mesure en :

- le nombre de lignes du code source écrites par programmeurs et par mois.
- Le nombre d'instructions du code objet produites par programmeur et par mois.
- Le nombre de pages de documentation écrites par programmeur et par mois.
- Le nombre de tests écrits et exécutés par programmeur et par mois

Etude comparative entre l'approche classique (CV) et l'approche prototypage
Selon une expérience faite dans la construction d'un système où certaines équipes ont adoptées l'approche traditionnelle (CV) et d'autres ont utilisée l'approche prototypage, les résultats obtenus sont les suivants :

- la taille des maquettes est plus petite que celle d'un produit fabriqué selon le cycle de vie (environ 40%)
- aucune différence sensible concernant la productivité (exprimée en lignes du code source/ personne/heure)

- les produits développés selon le cycle de vie sont plus fonctionnels, plus solides mais pas faciles à utiliser que les maquettes.

9.1.5 Conclusion

Le prototypage permet d'établir les conclusions suivantes :

- les produits sont plus petits, de performance équivalente avec moins d'effort.
- Les produits sont moins satisfaisants (aspect fonctionnel) mais plus simple d'emploi.
- Les produits sont simples à maintenir
- Les produits facilitent une meilleure utilisation du système
- Le système obtenu après prototypage est précis.

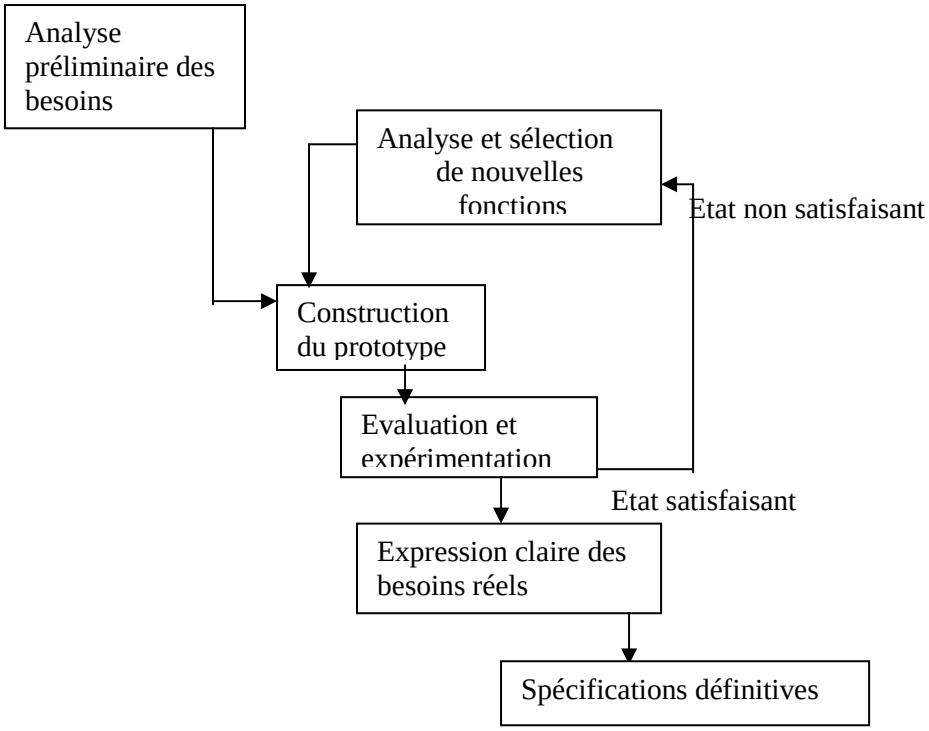
D'autre part, la comparaison entre une maquette jetable et un prototype évolutif donne les résultats suivants :

	maquette jetable	prototype évolutif
On obtient	rapide et provisoire	non rapide, non provisoire
On construit	uniquement les parties difficiles	d'abord les parties faciles ensuite difficiles
On gagne	un temps de développement optimal	une flexibilité optimale
Conclusion	jeter après usage	faire évoluer

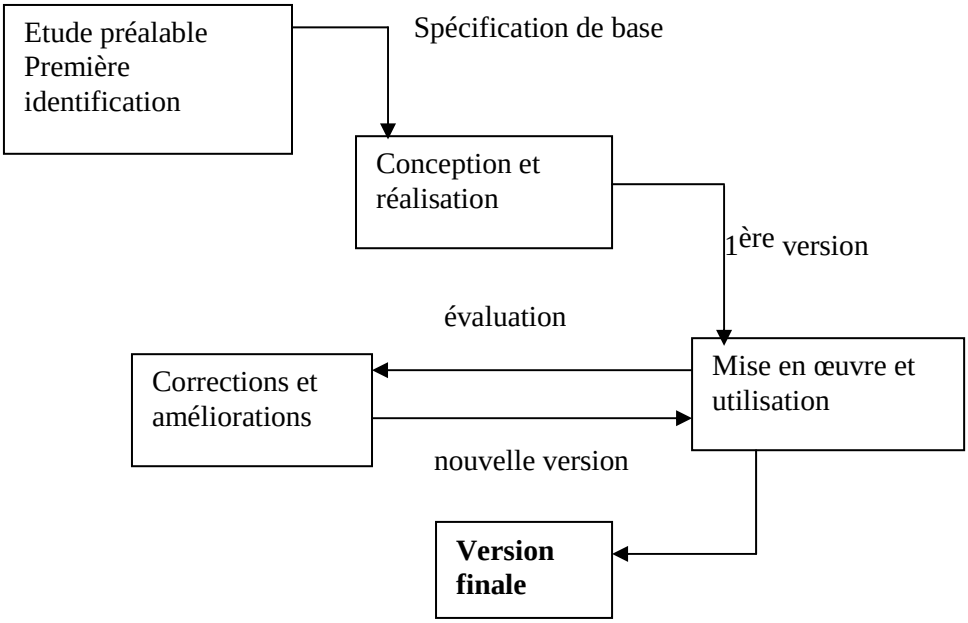
Exemples de prototypes

1. prototype rapide ou maquette utilisée en
   analyse des besoins
   spécifications fonctionnelles

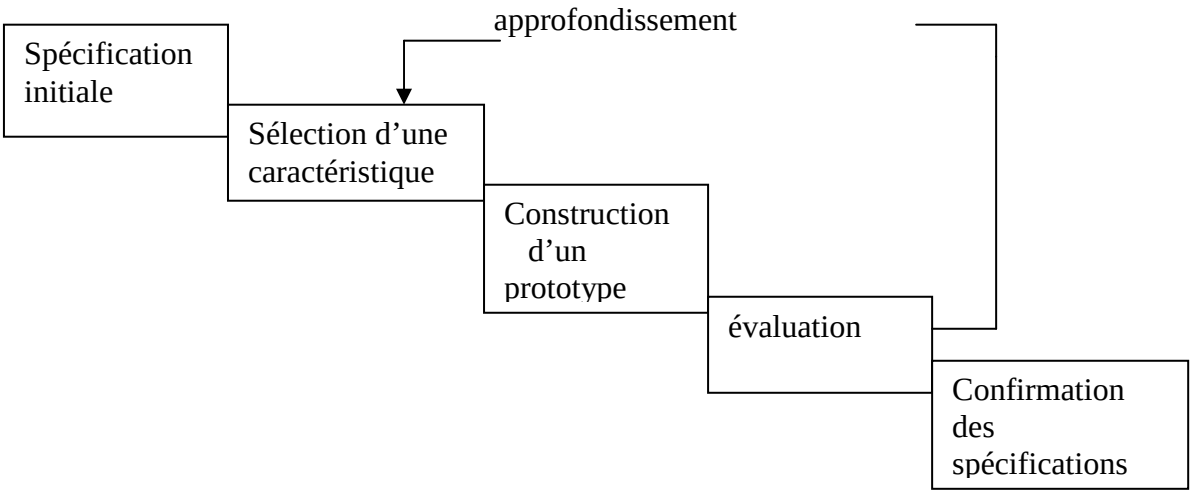
   Elle permet au client et au développeur de bien se mettre d'accord sur la nature du produit à réaliser et plus particulièrement sur l'interface et les fonctionnalités.



2. prototype évolutif
La première version du prototype est l'embryon du produit final. C'est un premier pas vers un produit final. On itère jusqu'au produit final.
Cas d'un système expert



3. prototype expérimentale
Généralement utilisé au niveau de la conception pour valider certaines propriétés ou options de conception.
Cas d'un analyseur syntaxique avec une grammaire réduite



Ce prototype est généralement jeté après développement.

9.2 Les réseaux de Pétri (RDP)

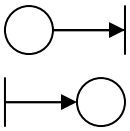
9.2.1 Introduction

- Thèse de Carl Adam Petri en 1962
- Le formalisme des réseaux de Pétri est un outil permettant l'étude des systèmes dynamiques et discrets.
- Il s'agit d'une représentation mathématique permettant la modélisation d'un système.
- L'analyse d'un réseau de Petri peut révéler des caractéristiques importantes du système concernant sa structure et son comportement.
- Les résultats de cette analyse servent à évaluer le système et en permettre la modification et/ou l'amélioration.

9.2.2. Les concepts de base

Un réseau de Petri est composé :

- un ensemble fini de places
- un ensemble fini de transitions
- un ensemble fini de flèches reliant :
  - soit une place à une transition
  - soit une transition à une place



9.2.3. Définition

Un réseau de Petri est un quadruplet (P, T, F, W) où

- P est un ensemble fini de place
- T est un ensemble fini de transitions
- F est la relation de flot (ce sont les flèches) F P*T T*P
- W est la fonction de poids qui associe un entier naturel non nul à chaque élément de F

  W : F $\longrightarrow$ N-{0}

  Si une flèche f de F n'est étiquetée d'un poids, par défaut W (f) = 1

Dans le cas où le poids affecté à chaque flèche est 1, le réseau est dit élémentaire.

W : F $\longrightarrow$ {1}

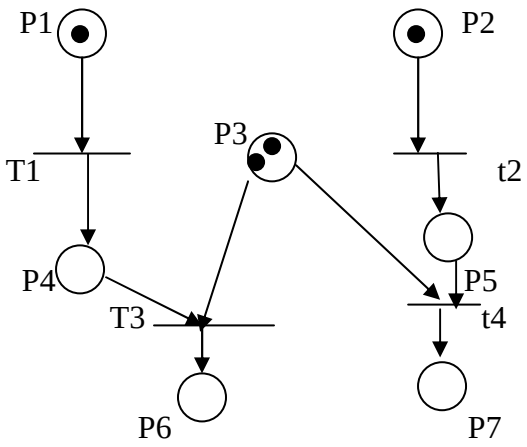
9.2.4. Le marquage d'un réseau de Petri

- on peut placer des jetons dans des places du réseau de Petri
- l'emplacement et la quantité des jetons définissent l'état du réseau de Petri à un instant donné
- On appelle marquage, la fonction qui décrit le nombre de jetons contenus dans chaque place

  M : P $\longrightarrow$ N

Exemple

M (p1) = 1
M (p2) = 1
M (p3) = 2
M (p4) = 0
M (p5) = 0
M (p6) = 0
M (p7) = 0



Un réseau marqué est un couple N = < R, P > où :
R est un réseau de Petri
M est un marquage c-a-d une application M : P ⟶ N / M (p) est le nombre de jetons dans la place p € P

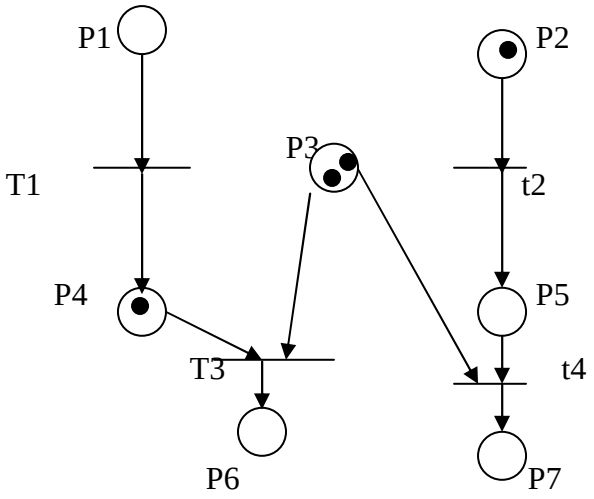
Un réseau sans marquage (aucun jeton dans chacune de ses places) est un réseau mort, sans activité.
Le marquage initial (noté M_0) est le premier marquage attribué à un réseau avant que toute ses transitions ne soient tirées.

9.2.5 Transition franchissable
Une transition est dite franchissable (tirable ou déclenchable) si chacune de ses places d'entrée contient un nombre de jetons égal ou supérieur à celui du poids affecté à la flèche reliant la place à la transition.

P est une place d'entrée et M (p) >= W (p, t)

Il suffit d'une seule place ne satisfaisant pas cette situation pour empêcher le franchissement.

Exemple
Quelles sont les transitions franchissables ? t2, t3

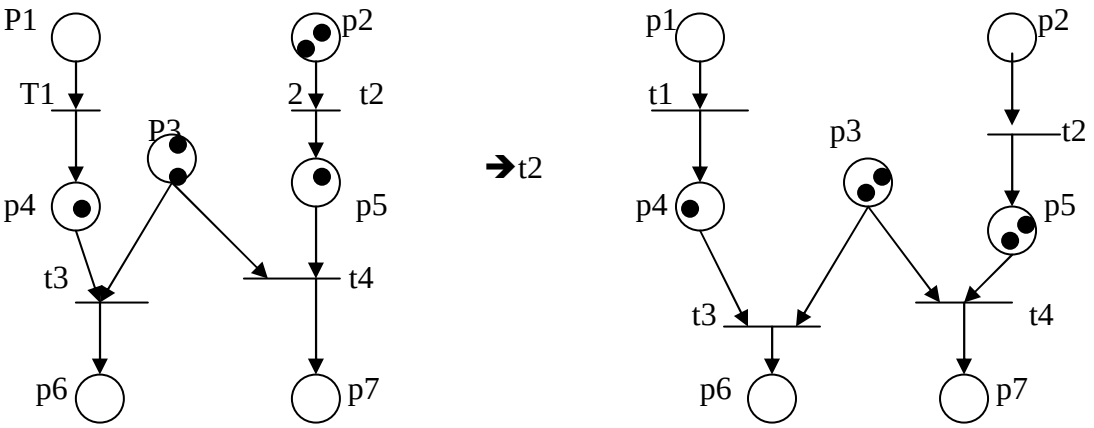


Une transition sans place d'entrée est toujours franchissable.

Franchissement d'une transition
Une transition franchissable peut être franchie (ou tirée) lorsqu'on franchit (ou on tire) une transition t :

- Pour chaque place d'entrée p_i un nombre de jeton égal au poids indiqué sur la flèche de p_i à t est retiré.
  Il y a consommation de jetons par la transition.
- Pour chaque place de sortie p_j, un nombre de jetons égal au poids indiqué sur la flèche de t à p_j est rajouté
  Il y a production de jetons par la transition.

La conservation du nombre global de jetons n'y est pas. Il peut y avoir 10 jetons consommés et 1 seul produit, 1 jeton consommé et 4 produits, etc.



Soient les fonctions Pré et post définissant les pré et post conditions du franchissement des transitions.

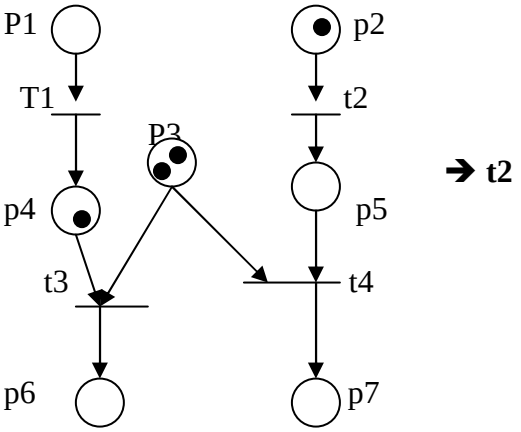
Pré(p, t) = w(p, t) si p est une place d'entrée
= 0 sinon
post(p, t) = w(t, p) si p est une place de sortie
= 0 sinon

Quelque soit p € P M'(p) = M(p) – pré(p, t) + post(p, t)

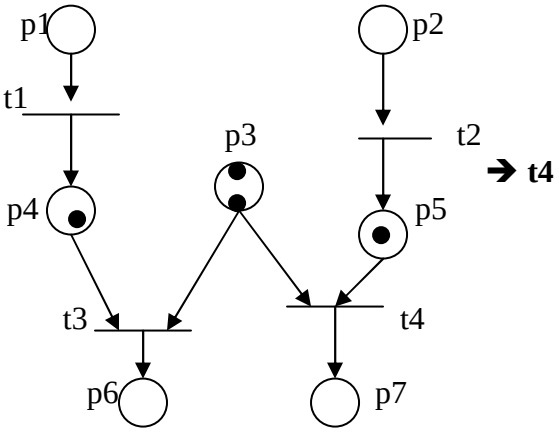
Exemple
Pour le franchissement de t2, on obtient M'

M(p1) = 0	pré(p1, t2) = 0	post(p1, t2) = 0	M'(p1) = 0
M(p2) = 2	pré(p2, t2) = 2	post (p2, t2) = 0	M'(p2) = 0
M(p3) = 2	pré(p3, t2) = 0	post (p3, t2) = 0	M'(p3) = 2
M(p4) = 1	pré(p4, t2) = 0	post (p4, t2) = 0	M'(p4) = 1
M(p5) = 1	pré(p5, t2) = 0	post (p5, t2) = 1	M'(p5) = 2
M(p6) = 0	pré(p6, t2) = 0	post (p6, t2) = 0	M'(p6) = 0
M(p7) = 0	pré(p7, t2) = 0	post (p7, t2) = 0	M'(p7) = 0

Séquence de franchissement
Soit un réseau marqué <R, M>, une séquence de franchissement est une suite de transitions franchies séquentiellement dans R depuis le marquage M
Notation < t1, t2, t3,……tn>
T1 est franchissable depuis le marquage M
T2 est franchissable depuis le marquage M' obtenu suite au franchissement de t1, etc.

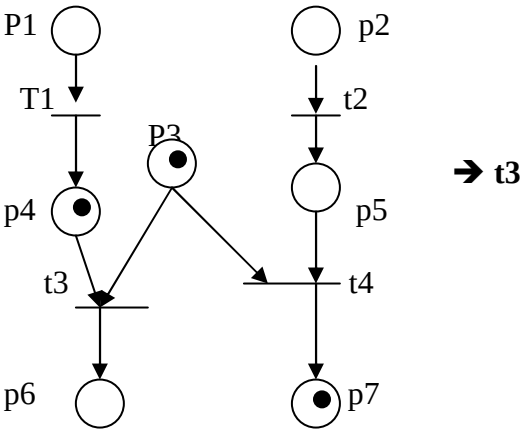


➔ **t2**

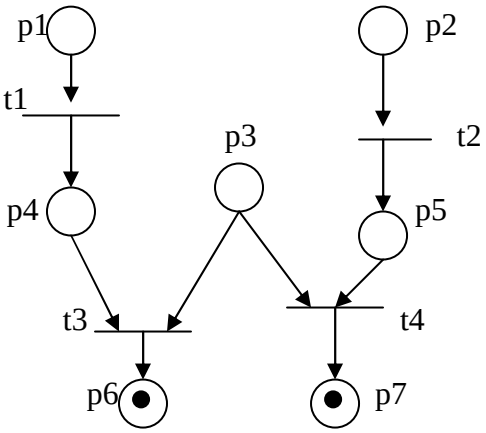


➔ **t4**

M <t2> M’



➔ **t3**



M <t2, t4> M’’ M < t2, t4, t3> M’’’

Notation

M<t1, t2, t3……..tn> ou $M \xrightarrow{t1, t2, t3 \ldots tn}$

La séquence <t1, t2, t3…tn> est franchissable depuis M

$M \xrightarrow{t1, t2, t3, \ldots \ tn} M'$

M<t1, t2, t3, tn> ou

La séquence <t1, t2, t3, …tn> est franchissable et conduit au marquage M’

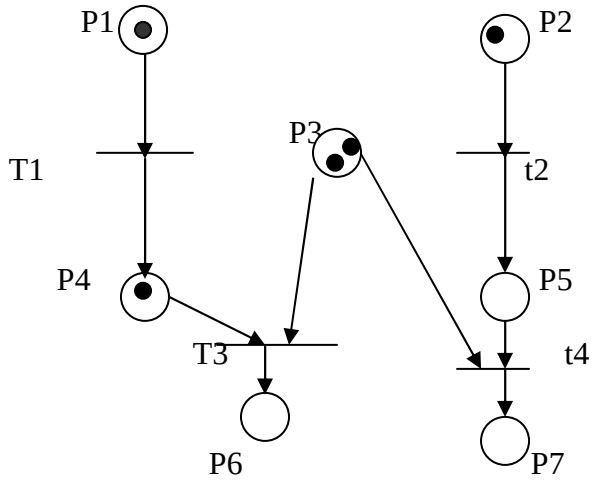
9.2.6. Propriété des réseaux de Petri

Les propriétés décrites ci-dessous concernent la structure du réseau et ne dépendent pas de la manière dont les jetons évoluent dans le réseau. Elles sont dites structurelles.

a) Non déterminisme

Le modèle des RDP est non déterministe si plus d’une transition est franchissable, le choix de la transition à franchir est non déterministe

t2 ou t3 ? choix non déterministe



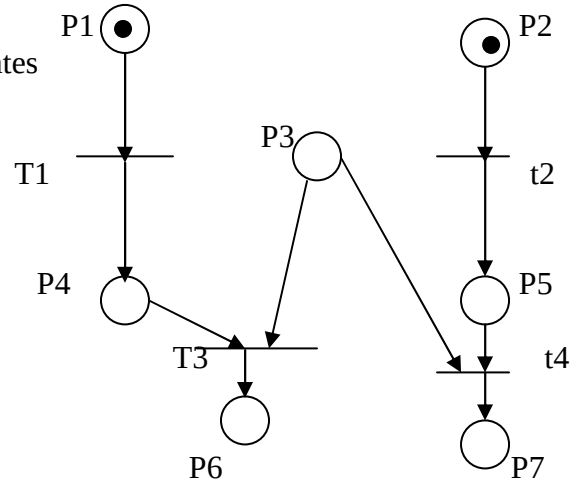
b) Concurrence

Transitions concurrentes : paires de transitions dont le franchissement de l'une n'empêche pas celui de l'autre.

Concurrence structurelle : deux transitions t1 et t2 sont structurellement concurrentes si elles n'ont aucune place d'entrée en commun

t1 et t2 sont structurellement concurrentes

T3, t4 ne le sont pas



c) Conflit

Transitions conflictuelles : le franchissement de l'une empêche le franchissement de l'autre.

Conflit structurel : deux transitions t1 et t2 sont structurellement en conflit si elles ont au moins une pace d'entrée en commun.
Dans l'exemple précédent t3 et t4 sont structurellement en conflit

d) Situation de blocage

Un RDP, dans un marquage donné, est dit en situation de blocage si et ssi aucune transition n'est franchissable.

9.2.7 Analyse des RDP

L'analyse des RDP peut permettre de confirmer ou infirmer certaines propriétés du système modélisé.

Outils : graphe de marquages accessibles

Graphe de couverture

Marquages accessibles

Etant donné <R, M_0>, l'ensemble des marquages accessibles est l'ensemble de tous les marquages de R atteignables par une séquence de franchissement à partir du marquage initial M_0.

A <R, M_0> = {M_i / il existe s $M_0 < s > M_i$

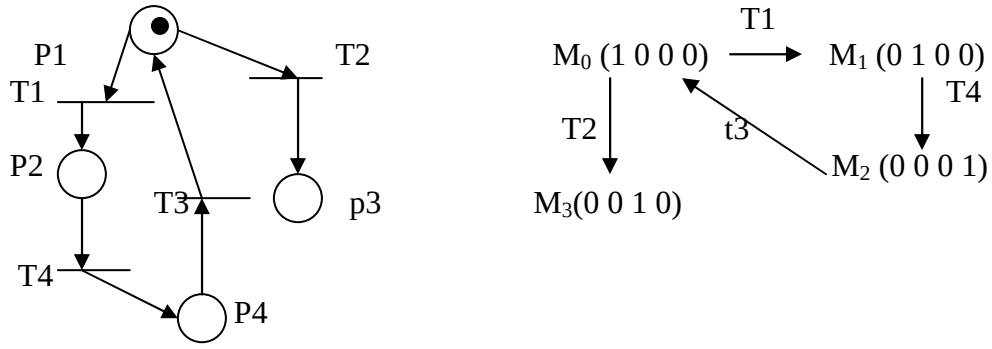
Graphe des marquages accessibles

Si A<R, M_0 > est fini ➔ possibilité d'une représentation sous forme d'un graphe.

Le graphe des marquages GA(R, M_0) est un graphe

- dont les sommets sont les marquages de A(R, M_0)
- tel qu'un arc relie deux sommets M_i et M_j s'il existe une transition franchissable permettant de passer du marquage M_i à M_j.

Exemple



Un algorithme assez simple est le suivant :

Tant qu' il y a des transitions franchissables faire

- Choisir une transition franchissable
- Franchir cette transition

Fin tant que

Remarque

Le graphe de marquage GA(R, M_0) ne peut être construit si l'ensemble des marquages est infini.

Exemple

Le cas d'une séquence faisant croître le nombre de jetons dans une place ➔ construction d'un graphe de couverture.

9.2.8. Propriétés comportementales

- Réversibilité :

Un RDP initialement marqué M_0 est dit réversible ssi :
Quelque soit M € A(R, M_0) il existe s / M < s > M_0
➔ retour au marquage initial

- K-borné

Le RDP initialement marqué M_0 est k-borné ssi :
Quelque soit M € A(R, M_0) M (p) <= k
➔ modélise les capacités des ressources.

- Vivacité

Un RDP initialement marqué M_0 est dit vivant ssi :
Quelque soit M € A(R, M_0) il existe s / M < s >
et ceci quelque soit t € T t € s
➔ Pas de blocage dans le système

9.2.9. Modélisation avec les Rdp

Plusieurs possibilités de modélisation avec les Rdp :

- description détaillée du processus de manipulation, de transformation de données.

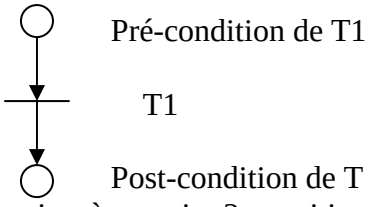
Ceci conduit à créer plusieurs Rdp, un par processus et à étudier la communication entre ces Rdp. Ensuite, il faut créer un seul Rdp global.
Exemple :
Dans le cas d'un logiciel de gestion des comptes-clients :
Dans une modélisation unitaire par processus : La description porte sur la vie d'un client ➔ inscription de ce client, ouverture de son compte, puis retraits et dépôts,….

- modélisation globale du système

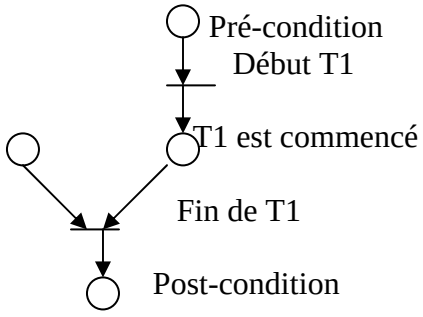
Dans ce cas, la description porte sur l'ensemble des clients ;
On procède par enregistrement aux retraits, pour continuer à recenser les mauvais payeurs et ensuite par le blocage de leurs comptes

a) Transitions- traitements et places - conditions

Transition : les traitements sont découpés en séquences d'opérations ininterruptibles ordonnées logiquement. A chaque séquence correspond une transition.



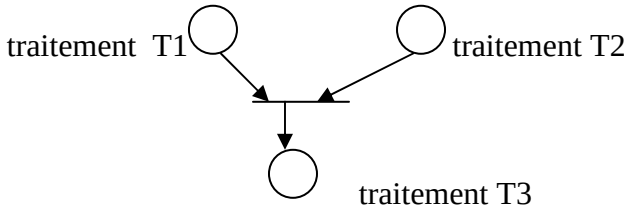
Une variante consiste à associer 2 transitions à chaque séquence, la première correspond au début du traitement et la seconde à la fin.



L’inconvénient est que le nombre de transitions est multiplié par 2.
Places : elle est assimilée à une variable logique pouvant prendre deux valeurs vraie et fausse.

b) Transitions- conditions et places traitements

Ce type de modélisation sert à visualiser l’ordre logique existant entre les traitements, les transitions servent à exprimer les liens de causalité entre les traitements.



Lorsque T1 et T2 seront terminés, T3 sera lancé/

c) Les jetons

- représenter un changement d’état, un seul jeton dans une place suffit.
- exprimer des quantités, autant de jetons qu’il faut

9.2.9. Conclusion

Les Rdps présentent peu d'éléments de base et donc sont simples d'emploi. Ils sont utilisés dans différents domaines tels que les protocoles de communication, les systèmes d'information, automatisme, etc. Cependant, certaines limitations du modèle de base sont à citer:

- absence de la notion de temps
- absence de la politique de sélection de transitions franchissables

➔ Des extensions ont été proposées à savoir : les réseaux de Petri colorés, les réseaux de Petri temporisés etc.

9.3. L'approche objet

9.3.1. Développement non objet

Deux principaux axes sont à distinguer dans le développent des systèmes :

a) approche fonctionnelle : pour certains problèmes, une fonction principale est à identifier et qui est affinée progressivement pour aboutir aux fonctions élémentaires selon une hiérarchie de décomposition. C'est une méthode intuitive mais elle nécessite de bien identifier les données et les contrôles échangés entre les diverses fonctions de la hiérarchie.

Des inconvénients sont liés à cette méthode à savoir :

- Certains systèmes ne possèdent pas de fonction globale.
- Maintenance difficile : un changement au niveau d'une fonction donnée nécessite de revoir tout le système.
- Faible réutilisabilité.

b) approche systémique : très utilisée pour développer des bases de données relationnelles classiques. Selon cette approche, le concepteur des BD doit avoir une vue synthétique de toutes les données manipulées par le système.

Comme un système possède trois vues : structurale, fonctionnelle et dynamique, seule la vision de structure est bien fournie, les deux autres visions ont une importance secondaire.

9.3.2 Nécessité d'un développement objet

- Les méthodologies classiques sont insuffisantes : les applications modernes sont importantes et complexes (temps réel + BD).
- Le problème de la réutilisation : les nouvelles applications ont besoin d'un certain nombre de procédures ou de fonctions déjà existantes avec peu de modifications.
- Application traversant les frontières de l'entreprise : pour des applications d'envergure, il est nécessaire d'avoir un modèle mettant en jeu plusieurs produits logiciels de plusieurs compagnies.
- Logiciels à composants : définir une façon simple, commode permettant aux usagers de fabriquer lui-même une application en associant tous les autres composants logiciels déjà disponibles.
- Développement en équipe : Le développement de grands logiciels nécessite une coopération de plusieurs équipes où chacune se charge de développer un module indépendamment des autres et de le tester séparément.
- Délai de mise en route des applications complexes et minimisation des coûts : l'utilisation de composants déjà existant dans la construction d'applications complexes permet de réduire le délai de mise en route ainsi que le coût.

9.3.3. Historique

En 1965, deux chercheurs norvégiens Ole Dahl et Kristen Nygaard ont proposés d'unifier les notions de procédures et de données grâce au mécanisme d'abstraction qu'ils nomment classes d'objets afin d'améliorer la capacité d'expression des langages traditionnels. Ils proposent le principe d'encapsulation, de classe, d'instance et de factorisation des propriétés des classes en graphe d'héritage. Ces principes ont donné lieu au langage Simula-67.

Le deuxième promoteur de ce mode de programmation fût Alan Kay qui introduit l'idée de communication par messages qui a donné lieu au langage Smallalk(1976).

En 1992, l'approche objet n'était qu'expérimentale dans beaucoup d'entreprises. Ce sont les applications client/serveur qui ont permis à la technologie objet de prendre place.

Plusieurs méthodes existent et dont l'unification des trois principales a donné naissance à UML.

9.3.4. Le concept de l'objet

L'intérêt de l'approche objet résulte de sa capacité de décrire les entités du monde réel et à les gérer sous forme naturelle dans laquelle les données et les procédures les manipulant sont regroupées dans une même entité sémantique : l'objet.

Un objet : élément identifiable du monde réel qui est soit concret ou abstrait.
Un objet est une entité identifiable possédant:

- un état
- un comportement

Exemple
objet : voiture

état	kilométrage vitesse immatriculation
comportement	rouler démarrer tomber en panne

Remarque
Le concept d'objet diffère de celui de l'entité par:

- l'entité ne s'intéresse qu'aux données (sa structure)
- l'objet englobe également les procédures de manipulation de ces données (comportement)

Une classe : regroupement d'objets ayants des propriétés communes (état et comportement).
Les objets membres d'une classe sont appelés instances de la classe.
Exemple
Classe « étudiants » : ensemble des étudiants du département d'informatique
« Samir » est une instance de la classe « étudiants »

9.3.5. Les principes sur lesquels reposent les objets

a) Encapsulation : regroupement des informations d'état et de comportement sous un nom unique.
Elle correspond à la fois à l'action

- de masquer à tout accès extérieur les propriétés de l'objet
- d'inclure dans l'objet les traitements et les données qu'il utilise.

Dans ce cas l'objet peut être réutilisé dans d'autres applications sans problèmes d'interférence.
L'encapsulation permet d'une part l'autonomie des objets et d'autre part la construction de bibliothèques de classes réutilisables.

b) masquage de l'information
On ne connaît pas la structure de l'information

c) Interface
Seuls les services publics (offerts à l'extérieur) sont utilisables.

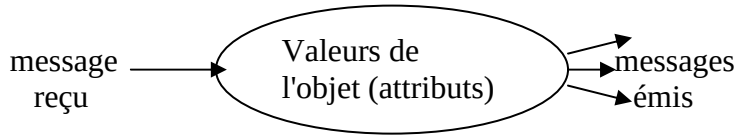
d) Envoi de messages
Les objets communiquent par messages (envoi / réception de messages).
La structure d'un message est la suivante:

M = (S, O, P) où

- S : sélecteur (nom de la méthode)
- O : objet receveur du message
- P : paramètres de la méthode

La réception d'un message par l'objet provoque l'exécution de la méthode invoquée.
Cette exécution peut

- changer l'état de l'objet receveur
- ou bien donner naissance à l'envoi d'autres messages



Objet

L'envoi de messages

e) Durée de vie

Elle dépend du contexte

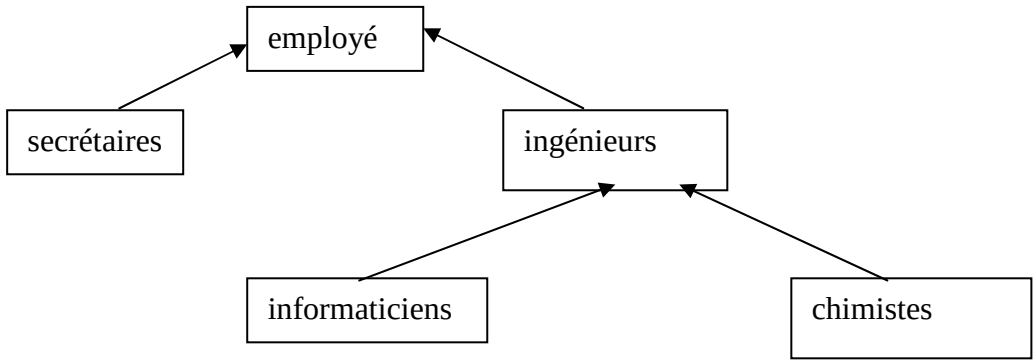
f) Polymorphisme

La forme de l'objet est différentes (mais cohérente) à des instants différents.

9.3.6. L'héritage

- permet la modélisation de l'emboîtement de classes les unes dans les autres la relation entre classe et sous-classe introduit des liens spécifiques appelés liens d'héritage.
- Ces liens mettent en évidence le transfert de propriétés (attributs et méthodes) de la classe mère vers ses sous-classes.

Exemple



Le lien d'héritage peut être parcouru dans les deux sens, d'où lien de généralisation / spécialisation.

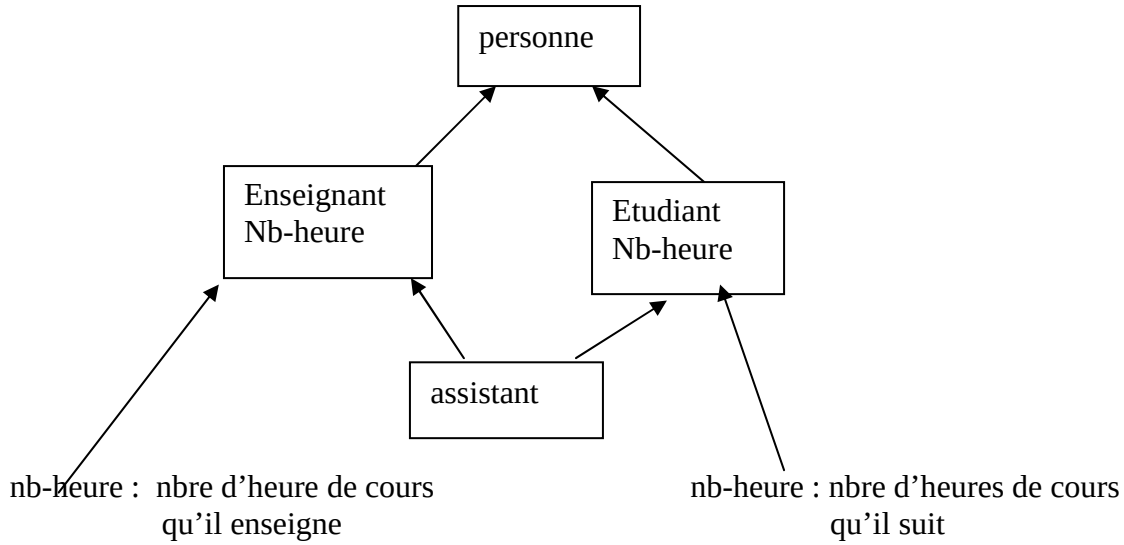
Types d'héritage

Il existe deux types d'héritage :

- l'héritage simple : lorsqu'une classe donnée ne peut hériter que d'une seule superclasse à la fois
- l'héritage multiple : lorsqu'une classe peut hériter de plusieurs classes à la fois.

Résolution de conflits
Les conflits de noms n'existent que dans le cas de l'héritage multiple.
- Dans le cas de l'héritage simple, il n'y a pas de conflit. Le graphe d'héritage est un arbre. Le mécanisme d'héritage consiste à rechercher dans la hiérarchie, du plus spécialisé au plus général, le premier ascendant possédant la propriété désirée.
- Dans le cas d'un héritage multiple, un objet peut avoir plusieurs ancêtres. Le parcours de l'arbre n'est pas naturel.

Exemple
Si l'objet A hérite de B et C alors
- définir une priorité en cas de conflit entre B et C pour une propriété donnée
- mentionner de façon explicite le nom de la superclasse à considérer en cas de conflit



La classe « assistant » hérite de toutes les propriétés
de la classe « enseignant » et de la classe « étudiant »

Le nombre d'heures « nb-heure » de l'assistant X dépend de la classe qu'il hérite.
D'où la nécessité de préciser l'attribut que l'on hérite en priorité.

9.3.7. Relation entre les objets
Trois types de relations avec des sémantiques différentes : la généralisation, l'agrégation et l'association
- Généralisation / spécialisation : la généralisation est une relation inverse de la spécialisation (relation entre classe et sous-classes)
- Composition / agrégation : L'agrégation permet de construire des objets complexes à partir d'autres objets appelés objets composants (relation composé - composants).
- L'association : mécanisme regroupant des objets qui ne sont pas reliés par une relation de généralisation / spécialisation ou une relation de composition / agrégation. C'est une association des objets du monde réel reliés par une activité.

9.3.8. Conclusion

Intérêt des objets:

- Maîtrise de la complexité:
  L'encapsulation permet de se concentrer sur un objet et un seul ou sur une classe et une seule. Il possible de tester un objet de façon presque totale.
- Evolution facilitée : rajouter de nouveaux objets est simplifié ➔ on raccroche aux modèles déjà construits par généralisation / spécialisation.
- Réutilisation : très possible à cause de la relation de spécialisation / généralisation.

Inconvénients de l'approche orientée objet

- Tester un système orienté objet est difficile.
- Penser objet oblige à changer de mentalité.

Le génie logiciel

Le but de ce chapitre est de présenter des définitions de certaines notions de base en génie logiciel. Il est organisé comme suit:

1. Historique
2. Définition d'un logiciel
3. La crise du logiciel
4. Le génie logiciel
5. La qualité du logiciel
6. Le cycle de vie d'un logiciel
7. Les ateliers de génie logiciel

1. Historique

- Au début (années 50), il y avait de la programmation:
  - petits programmes sur mesure, de type batch avec un seul client le plus souvent le client est aussi le programmeur
  - la naissance de la profession de programmeur.
- Années 60 et 70, les premiers « gros logiciels » apparaissent
  - les produits logiciels sont vendus à plusieurs clients (milliers)
  - des logiciels multi utilisateurs, temps réel
  - apparition des bases de données
  - crise du logiciel
- Depuis les années 80, l'industrie du logiciel se développe en même temps que les ordinateurs personnels, les réseaux, l'Internet, etc.
  - le coût des ordinateurs a baissé. Beaucoup d'ordinateurs sont vendus ainsi que des logiciels, des réseaux et des systèmes répartis, et la crise continue.

2. Définition d'un logiciel

Qu'est-ce qu'un logiciel?

Un logiciel est un ensemble

de programmes,
procédés,
règles et éventuellement de la documentation

relatifs au fonctionnement d'un ensemble de traitement de l'information (arrêté du 22 déc. 81).

Autrement dit:

Un logiciel est un ensemble de programmes informatiques mais aussi un certain nombre de documents nécessaire à leur installation, utilisation, développement et maintenance: spécifications, jeux d'essai, schémas conceptuels, mode d'emploi.

3. La crise du logiciel

Elle est apparue à la fin des années 60. Elle résulte du décalage entre les progrès matériels et logiciels.

D'une part, les ordinateurs de la troisième génération apparaissaient et qui étaient de plus en plus puissants et de moins en moins coûteux et d'autre part, la construction des logiciels restait artisanale.

Les nouvelles machines rendaient toutes applications (irréalisables) possibles alors que les méthodes de développement logiciel ne s'appliquaient pas à de grands systèmes.

Les symptômes les plus caractéristiques de cette crise sont les suivantes:

- Les logiciels réalisés ne correspondent souvent pas aux besoins des utilisateurs.
- Les logiciels contiennent trop d'erreurs (qualité du logiciel insuffisante)
- Les coûts de développement sont rarement prévisibles
- La maintenance des logiciels est une tâche complexe et coûteuse.
- Les délais de réalisation sont généralement dépassés.
- Les logiciels sont rarement portables.

Un certain nombre de mythes est à rapprocher de ces symptômes:

- Une idée grossière du logiciel à réaliser est suffisante pour commencer d'écrire un programme.
  Faux: une idée imprécise du logiciel à réaliser est la cause principale d'échecs.
- Une fois que le programme est écrit et fonctionne, le travail est terminé.
  Faux: la maintenance du logiciel représente un travail important, le coût de la maintenance représente d'ailleurs plus de 50% du coût total du logiciel.
- Si les spécifications du logiciel à réaliser changent continuellement, cela ne pose pas de problèmes puisque le logiciel est un produit souple
  Faux : des changements de spécifications peuvent se révéler coûteux et prolonger les délais
- Pour respecter les délais prévus à la réalisation d'un logiciel, il suffit d'ajouter plus de programmeurs.
  Faux: si l'on ajoute des gens à un projet, une période de familiarisation est nécessaire. Le temps passé à communiquer à l'intérieur du groupe augmente lorsque la taille du groupe augmente. Cela réduit la productivité de chacun. L'introduction de nouvelles personnes dans le projet doivent être étudiée et planifiée soigneusement.
- Un équipement informatique à la fine pointe de la technologie ne garantit en rien une production logicielle de qualité.
  Vrai: la qualité est assurée en respectant les étapes du cycle de vie.

4. Le génie logiciel

Le génie logiciel est la discipline née en réponse à la crise du logiciel. Il se caractérise par une approche rigoureuse et systématique pour la construction de logiciels fiables, efficaces, peu coûteux et dans des délais raisonnables.

Le génie logiciel se préoccupe des procédés de fabrication des logiciels de façon que les quatre (4) critères suivants soient satisfaits:

- Le système / le logiciel qui est fabriqué répond aux besoins des utilisateurs.
- La qualité correspond au contrat de service initial. Elle regroupe plusieurs facteurs tels que l'extensibilité, la fiabilité, la compatibilité, la réutilisabilité, l'intégrité, etc.
- Les coûts restent dans les limites prévus au départ.
- Les délais restent dans les limites prévus au départ.

Règle du CQFD : Coût Qualité Fonctionnalités Délai

4.1 Quelques définitions possibles

a) Le génie logiciel est donc l'art
de spécifier,
de concevoir,
de réaliser et de faire évoluer,
avec des moyens et dans des délais raisonnables, des programmes, des documentations et des procédures de qualité en vu d'utiliser un système informatique pour résoudre certains problèmes.

b) Le génie logiciel c'est l'application de méthodes scientifiques au développement
de théories,
de méthodes,
de techniques,
de langages et d'outils
favorisant la production de logiciel de qualité.

c) Le génie logiciel est une discipline de l'informatique qui regroupe un ensemble
de connaissances,
de procédés,
des acquis scientifiques
pour la conception,
la vérification et la documentation
du logiciel dans le but d'en optimiser la production, le support et la qualité.

Autrement dit:
Le génie logiciel est l'art de produire de bons logiciels avec :
un meilleur rapport **qualité / prix.**

5. Qualité du logiciel

En génie logiciel, la définition de la qualité du logiciel est exprimée en termes de facteurs.
Ces facteurs peuvent être
- internes : visibles, observables par les développeurs
- externes : visibles par les utilisateurs. Parmi ces derniers, citons:

- Validité: aptitudes d'un produit logiciel à remplir exactement ses fonctions, définies par le cahier des charges et les spécifications
- Extensibilité: facilité avec laquelle un logiciel se prête à une modification ou à une extension des fonctions qui lui sont demandées.
- Compatibilité: facilité avec laquelle un logiciel peut être combiné avec d'autres logiciels.
- Efficacité: utilisation optimale des ressources matérielles.
- Portabilité: facilité avec laquelle un logiciel peut être transférée sous différents environnements matériels et logiciels.
- Intégrité: aptitude d'un logiciel à protéger son code et ses données contre des accès non autorisés.
- Facilité d'emploi: facilité d'apprentissage, d'utilisation, de préparation des données, d'interprétation des erreurs et de rattrapage en cas d'erreur d'utilisation.

- Réutilisabilité: aptitude d'un logiciel à être réutilisé, en tout ou en partie, dans de nouvelles applications.
- Robustesse : aptitude d'un logiciel à continuer à s'exécuter en dépit de certaines violations des suppositions dans les spécifications.

Les facteurs internes influencent directement les qualités externes.
Exemple
Le logiciel doit être vérifiable (interne) pour pouvoir affirmer qu'il est fiable (externe).
Les facteurs externes peuvent entretenir des relations entre eux.
Exemple:
Maintenabilité ➔ compréhensibilité.

6. Cycle de vie d'un logiciel

6.1 Définition (IEEE)

La production d'un logiciel comprend plusieurs étapes et s'étend sur une longue période de temps. Le cycle de vie d'un logiciel: c'est la période de temps qui débute au moment de la définition et du développement du produit logiciel et se termine lorsque le produit logiciel n'est plus disponible.
Le but de ce découpage est
- maîtriser les risques
- maîtriser les délais et les coûts d'une façon plus meilleure.
- assurer la qualité attendue.

6.2 Justification du cycle de vie

La relation entre le cycle de vie et l'assurance qualité est très forte. Il s'agit d'assurer en permanence :
- la validation : est ce qu'on est en train de faire le bon produit?
- la vérification : est ce qu'on fait le produit correctement?

On parle de "V& V" (validation et vérification)
Comment?
La validation et la vérification sont garanties par la mise en place d'inspections et de revues.
L'inspection est une lecture critique d'un document (de spécification, de conception, de code, de plan d'intégration, etc.) afin d'améliorer sa qualité. Elle est effectuée par une équipe indépendante du projet et comprenant le client éventuellement.
L'inspection donne lieu à la rédaction de fiches de défauts précisant le degré de gravité et la définition des responsabilités concernant la correction des défauts.
Les inspections sont effectuées à la suite des décisions prises en revue.
Une revue est une réunion permettant de valider une des phases du cycle de vie.
Parmi ces revues, on cite:
Les revues techniques qui permettent de fournir au marketing et à l'unité de développement une évaluation des aspects techniques du projet et des coûts de réalisation.

Plus une erreur est découverte tard dans le cycle de vie, plus la réparation est coûteuse.

6.3 Les étapes du cycle de vie

L'étude préalable: elle consiste à élaborer une version de base du cahier de charges qui doit inclure

- la décision de faisabilité
- le plan général du projet
- une estimation approchée du coût et des délais de réalisation

Les concepteurs doivent effectuer une enquête précise au niveau des demandeurs de l'application (les clients) et des utilisateurs (souvent le personnel des clients) et une étude précise du système d'information.

La spécification: elle a pour but:

- la description du modèle fonctionnel du système d'information (quels sont les modules constitutifs? Comment l'ensemble fonctionne.)
- l'estimation des ressources (humaines et matérielles)
- une première version de l'échéancier de développement.

Les concepteurs vont s'appuyer sur les résultats de l'enquête des demandeurs et utilisateurs ainsi que sur des techniques de planification (recherche opérationnelle).
Et ceci va permettre d'obtenir un cahier de charges détaillé (fonctionnalités, ressources, planning).

La conception générale: elle consiste en une description du logiciel c-a-d obtenir à la fois une description du système d'information en un ensemble de modules et structures de données et une description du rôle de chaque module individuellement et aussi en interaction avec les autres.

La conception détaillée: elle consiste à obtenir une description détaillée des traitements des structures de données (algorithmes). La description doit faire apparaître de façon précise les interfaces (communication) entre les différents modules.

La codification (ou codage): elle permet de traduire chaque algorithme en un ou plusieurs langages de programmation mais aussi la rédaction de la documentation technique.

Les tests unitaires: ils consistent à vérifier le bon fonctionnement du programme et des différents modules le composant par l'intermédiaire de jeu d'essai bien choisi au sein du domaine d'application

Les tests système: ils correspondent

- à la phase d'intégration des différents programmes pour obtenir le logiciel fini.
- ainsi qu'à l'ensemble des tests qui permettra de vérifier que le logiciel correspond exactement au cahier des charges
- enfin à la rédaction de la documentation utilisateur.

L'exploitation: elle consiste à la mise à disposition de tout utilisateur du logiciel fini de tel façon à ce que ce logiciel soit opérationnel en situation réelle de production.

La maintenance qui peut être:

- Corrective : permet de corriger les erreurs qui n'ont pas été détectées lors des phases précédentes de test.
- adaptative : doit s'occuper de faire évoluer et adapter le logiciel à l'apparition de nouvelles contraintes.
- perfective: a pour objectifs l'optimisation des performances du logiciel.

Cette catégorisation peut être étendue pour regrouper d'autres catégories (évolutive , préventive)

6.4 Les différents cycles de vie

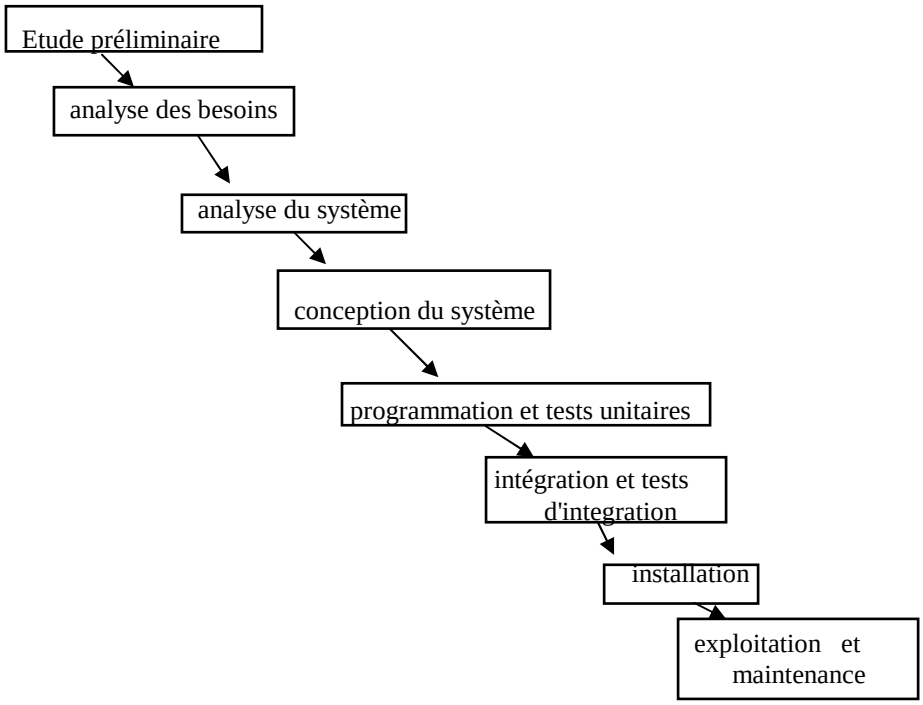
On distingue deux grandes classes:

- séquentiels
- itératifs

6.4.1. Les cycles séquentiels

- Le modèle de la cascade

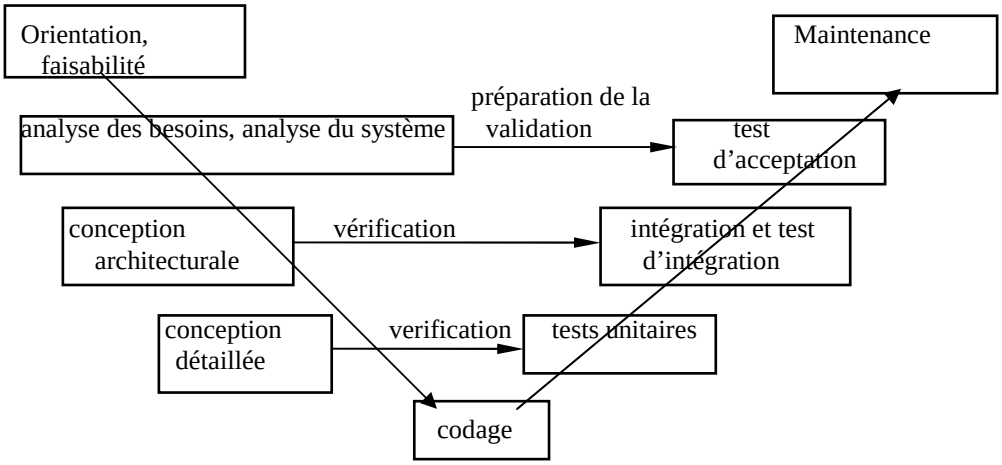
Le principe du modèle de la cascade (ou modèle de la chute d'eau) est très simple. Il se présente comme une succession d'étapes correspondant à une approche descendante.
Une étape doit se terminer à une certaine date, par la production de certains documents ou logiciels. Ces résultats sont soumis à une revue approfondie et le passage à l'étape suivante est possible que s'ils sont jugés satisfaisants.



Dans sa première version, le modèle ne comporte que les flèches descendantes, qui désignent l'enchaînement des étapes. Les flèches ascendantes qui ont été rajoutées expriment le principe qu'une étape ne remet en cause que l'étape précédente.

- Le modèle en V

Le principe de ce modèle est que toute décomposition doit être décrite avec la recomposition et chaque description d'un composant est accompagnée de tests qui permettront de s'assurer qu'il correspond à sa description.



Il existe deux sortes de dépendances entre les étapes:

- Celles qui désignent que les étapes se déroulent de manière séquentielle en suivant le V de gauche à droite.
- Celles qui représentent le fait qu'une partie des résultats de l'étape de départ est utilisée directement dans l'étape d'arrivée.

Exemple

A l'issue de la conception architecturale, le protocole d'intégration et les jeux de test d'intégration doivent être complètement décrits.

Critiques des modèles en V et en cascade:

a) Modèles parfois difficiles à appliquer car il est difficile de prendre en compte des changements importants dans les spécifications dans une phase avancée du projet.

b) Gestion des risques. Trop de choses sont reportées à l'étape de programmation (interface utilisateur)

6.4.2 Les cycles itératifs

- Le modèle en spirale

Ce modèle, proposé par B. Boehm en 1988, est plus général que les précédents et peut les inclure. Il met l'accent sur l'analyse des risques.

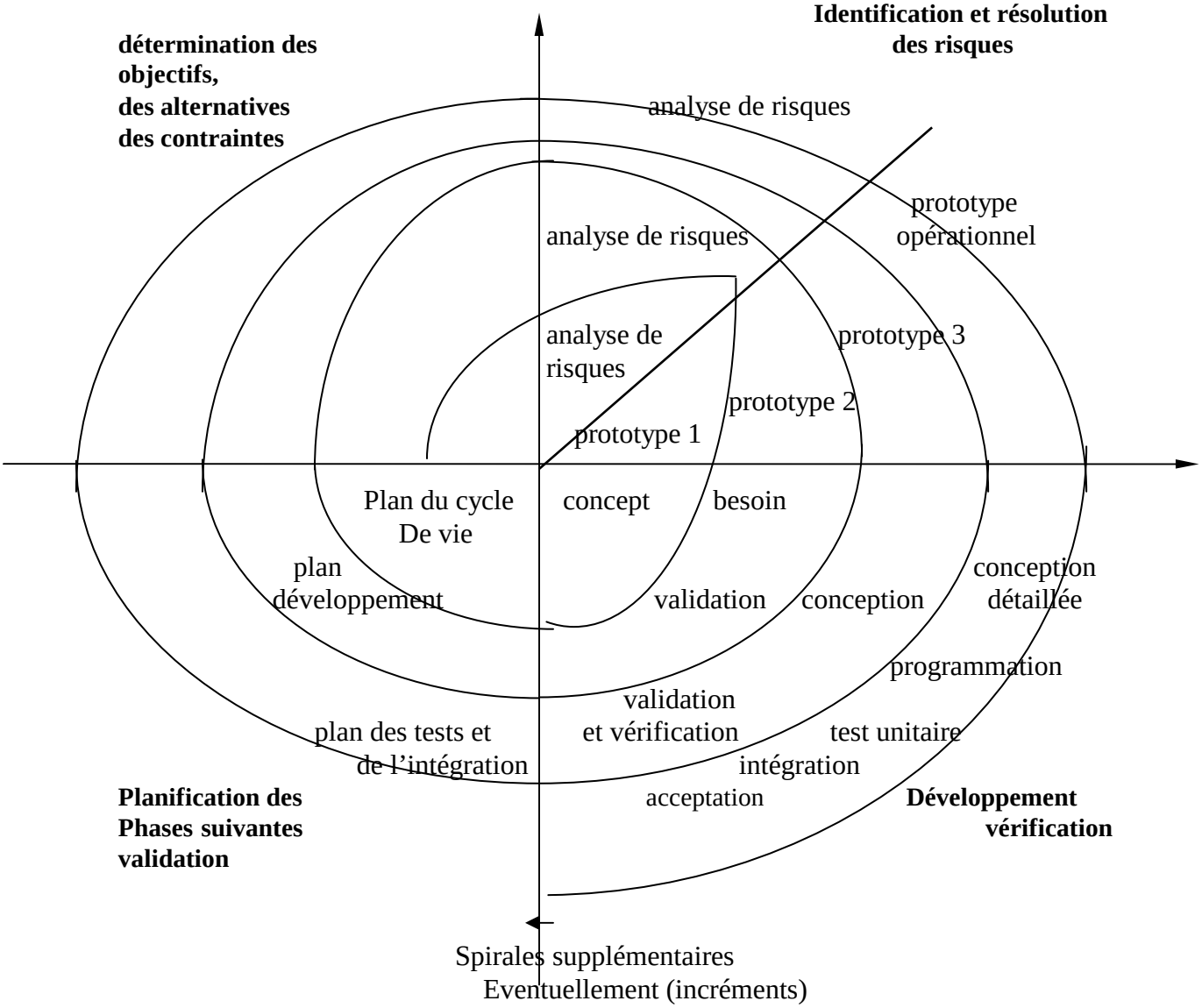
Le principe est le suivant:

- Identifier les risques, leur affecter une priorité.
- Développer des prototypes pour réduire les risques en commençant par le plus grand risque.
- Utiliser un modèle en V ou en cascade pour implémenter chaque cycle de développement
- Contrôler

  Si un cycle concernant un risque est achevé avec succès, évaluer le résultat du cycle et planifier le cycle suivant.

  Si le risque est non résolu, interrompre le projet.

Le nombre de cycles est variable selon que le développement est classique ou incrémental.



Voici quelques types de risques:

Risques humains

- Défaillance de personnel
- Travailleur solitaire
- Surestimation des compétences.

Risques liés au processus

- Calendrier et budgets irréalistes.
- Insuffisance des données.
- Développement des fonctions inappropriées.
- Développement d'interfaces utilisateurs inappropriées.

Risques technologiques

- Changement de technologie en cours de route.
- Incompréhension des fondements de la technologie

Caractéristiques du modèle en spirale

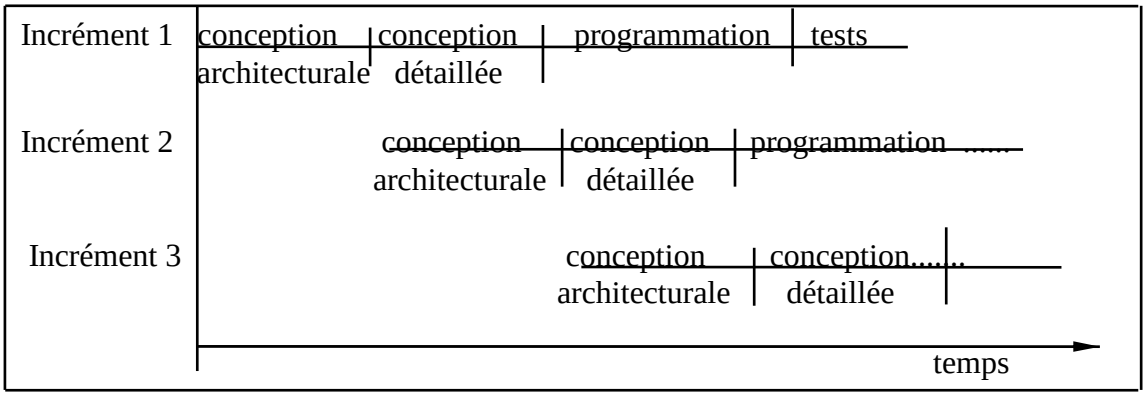
- Utilisation du prototypage.
- Analyse des risques.

- Le modèle par incréments

Dans ces modèles, un logiciel noyau est d'abord développé puis des incréments sont successivement développés et intégrés.
L'itération porte sur la conception - le codage ou la programmation – le test.
Les avantages de ce modèle sont:

- Chaque développement est moins complexe
- Les intégrations sont progressives

Il peut y avoir des livraisons et des mises en service après chaque intégration d'incréments. Cette approche est souvent utilisée pour de grands projets faisant appels à des sous-traitances.



- Avec prototypes éventuellement (voir exemple1, prototype rapide)

L'itération porte sur

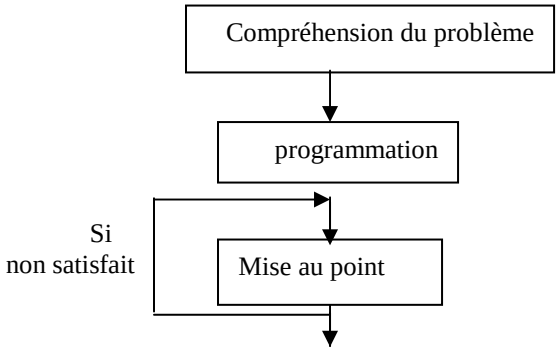
- expression des besoins
- spécification et prototypage

On passe à la conception globale quand les besoins et les spécifications ont été validés.

6.4.3 Autres cycles de vie

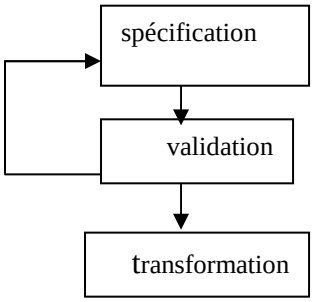
- Le modèle du code-and-fix

Après une brève compréhension de l'objectif, l'application est développée. Plusieurs cycles de mise au point, parfois avec l'utilisateur du futur système, permettent d'atteindre le résultat visé.



- Le modèle da la transformation automatique

Il est basé sur la possibilité de transformer automatiquement les spécifications en programmes. Une succession de cycles de spécification/validation s'achève par la génération de code.



Le choix d'un modèle dépend de certains facteurs à savoir:

- Le type de projet,
- l'équipe chargée de sa réalisation,
- nature du projet (logiciel sur mesure, produit grand public),
- les contraintes du client.

7. Les ateliers de génie logiciel

La mise en place d'une organisation industrielle de production et de maintenance de logiciels doit être supportée par des AGL (Atelier de Génie Logiciel) ou atelier CASE (Computer Aided Software Engineering).

Un AGL regroupe :

- les méthodes reconnues : SADT, SA-RT, etc.
- Les procédures de développement normalisées (DoD2165, etc.)
- Les générateurs de codes PASCAL, C, ADA, etc.
- Les outils de génie logiciel tels que modèle d'estimation de coût.

7.1 Principaux objectifs des A.G.L

- Augmenter la productivité d'une équipe de développement.
- Améliorer la qualité des produits logiciels (maintenabilité), fiabilité, etc.)
- Aider l'équipe à appliquer les différentes normes, procédures dans le processus de développement.
- Soulager l'équipe des tâches répétitives (vérification de cohérence) lors des phases de spécification et de conception de logiciel.

7.2 Les différents types d'A.G.L

On distingue deux types d'A.G.L selon la nature des outils intégrés:

- Les environnements de conception (upper case)

Ces ateliers s'intéressent plus particulièrement aux phases d'analyse et de conception du processus logiciel.

Ils intègrent des outils pour l'édition

- des diagrammes,
- des dictionnaires de données, etc.

Ils sont basés sur une méthode d'analyse et de conception (MERISE).

- Les environnements de développement (lower case)

Ils s'intéressent plus particulièrement aux phases d'implémentation et de test du processus logiciel.

Ils intègrent

- des générateurs d'interface homme / machine (IHM),
- des SGBD,
- des compilateurs.

Expression et analyse des besoins

Le but de ce chapitre est de décrire la première étape du cycle de vie du logiciel : L'expression des besoins suivi d'un aperçu sur l'analyse/la spécification des besoins. Il est organisé comme suit:

1. Introduction
2. Les problèmes potentiels
3. Qu'est ce qu'un besoin?
4. Processus d'analyse des besoins
5. Conclusion

1. Introduction

L'analyse des besoins est :

- Une activité essentielle au début du processus de développement.
- Une activité dont les données sont fournies par des experts du domaine d'application et les futurs utilisateurs.
- Son rôle est d'éviter de développer un logiciel non adéquat.
- Les résultats de cette activité sont un ensemble de documents décrivant les aspects du futur système, son rôle et sa future utilisation.

2. Les problèmes potentiels

Deux issues sont à considérer :

- Le client : a une idée vague (des éléments manquants) ou change souvent d'idées.
- L'équipe de développement : les détails restent en dernier conduit à des erreurs de spécification et de développement.

3. Qu'est ce qu'un besoin ?

Un besoin (requis, exigence) est une exigence que le système doit satisfaire.

Catégories de besoins

- Besoins fonctionnels :
  - description des services (fonctions)
  - description des données manipulées.
- Besoins non fonctionnels :
  - description des contraintes.
  - Pour chaque service et pour le système, il est possible d'exprimer différents types de contraintes (contraintes de performance, de sécurité, etc.)

Les besoins non fonctionnels sont souvent appelés les attributs de qualité.

Ils spécifient comment le système accomplit bien ses fonctions:

Comment doit-il répondre rapidement?

Comment doit-il être utilisé facilement?

Comment il doit être protégé contre les attaques?

Comment il doit être maintenu ou mis à jour facilement?

Caractéristiques des besoins

- Corrects
- Clairs, sans ambiguïtés
- Cohérents
- Complets
- Réalistes

- Vérifiables
- Traçables

4. Processus d'analyse des besoins

- Expression des besoins
  - Participants : analyste, client et utilisateurs.
  - Document : cahier de charges, rédigé par le client en collaboration avec l'analyste en langage naturel.
- Spécification de besoin
  - participant : analyste.
  - document : dossier d'analyse et de spécification, rédigé par l'analyste en utilisant une notation graphique (modèles)

Expression des besoins

Détermination des besoins

Méthodes traditionnelles :

- Entrevues avec le client et les experts du domaine.
- Questionnaires.
- Observation.
- Etude des documents et des systèmes existants.

Méthodes actuelles :

- Prototypage (prototype jetable ou évolutif).
- Cas d'utilisation : description des scénario d'utilisation du logiciel
  a) Identification des services (cas d'utilisation) offert par le système.
  b) Identification des acteurs participant à chacun des cas d'utilisation.
  c) Description détaillée des scénarios d'exécution de chaque cas d'utilisation.

4.1.2 Validation et négociation

Les besoins répondent-ils aux exigences du client ? Il faut réviser la liste des besoins en vérifiant s'ils sont complets, clairs, etc. Ensuite, il faut les classer selon leur priorité et évaluer les risques associés à chacun.

Eliminer les besoins irréalistes ou non pertinents

- Bien délimiter les frontières du système : construire le diagramme de contexte et les entités externes, les entrées, les sorties puis identifier les besoins qui ne répondent pas aux objectifs du système.

Evaluer les risques associés aux besoins et évaluer leur priorité

- Quels sont les besoins qui peuvent causer des problèmes pendant le développement ?
- Quelle est la priorité attribuée à chaque besoin ? essentiel, utile, difficile, etc.

4.1.3 Gestion des besoins

Identification et classification des besoins dans un cahier de charges.

- identificateur unique.
- numérotation séquentielle.

Hiérarchisation des besoins

- un besoin peut se composer en un ou plusieurs sous besoins plus spécifique

Gestion des modifications et traçabilité

Lorsqu'une exigence est changée, comment retracer facilement les documents, modèles et les morceaux de code à modifier ?
utilisation d'un outil de gestion de configuration

4.1.4 Cahier des charges

Un cahier de charges a pour but d'établir une description globale des besoins d'un nouveau produit ou des extensions d'un produit existant à partir de la spécification des objectifs. Il est généralement exprimé en langue naturelle d'où des problèmes d'ambiguïté et de consistance se posent. Cependant, l'avantage réside dans le fait qu'il est facilement compréhensible par le client.

4.1.4.1 Caractéristiques d'un cahier de charges

Un bon cahier de charges doit posséder les 7 qualificatifs suivants :

- Non ambigu : La non ambiguïté exige une grande précision dans l'utilisation des termes introduits.
- Complet : il faut préciser le comportement du logiciel lors des évènements non désirés (panne du matériel, erreur dans les données introduites par l'utilisateur)
- Vérifiable : toute spécification doit pouvoir être vérifiable.
  Un exemple de spécification non vérifiable est : « le logiciel doit être facile à utiliser »
- Consistant : pas de spécification contradictoire. Ce problème apparaît à partir d'une certaine taille du cahier de charges.
- Modifiable : les spécifications doivent changer soit durant le développement du logiciel soit durant la maintenance. Ces modifications doivent pouvoir être reportées facilement dans le cahier de charges.
- Traçable : le traçage est la possibilité d'avoir des références croisées entre les spécifications de plusieurs versions du cahier de charges.
  - Traçage arrière : à partir d'une spécification, retrouver la spécification dont elle découle.
  - Traçage avant : à partir d'une spécification, retrouver les spécifications auxquelles elle a donné naissance (dans les versions suivantes).
- Utilisable durant la phase de maintenance : un cahier de charges doit prévoir certaines évolutions du logiciel.

4.1.4.2 La structure d'un cahier de charges

Elle se présente comme suit :

a) Introduction : présentation du logiciel en termes de besoins (à quoi va-t-il servir), présentation des notations, des interfaces.
b) Matériel : description du matériel utilisé (les périphériques) ainsi que les interfaces.
c) Modèle conceptuel : vue d'ensemble du logiciel.
d) Besoins fonctionnels : description des fonctions (opérations, transformations) que le logiciel doit réaliser. Chaque fonction sera décrite en détail en spécifiant ses entrées et ses sorties. Les valeurs légales pour les entrées ainsi que le comportement du programme pour ces entrées est aussi spécifié.

e) Besoins non fonctionnels : ce sont les contraintes qui sont décrites.
Elles sont de deux types :
Les contraintes d'interface où l'on distingue :
1. Les contraintes imposées par l'environnement logiciel (exécution du programme sur tel système d'exploitation)
2. Les contraintes imposées par l'environnement matériel (utilisation des caractéristiques de tel terminal)
3. Les contraintes imposées par l'environnement humain (les commandes mises à disposition doivent satisfaire telle ou telle contrainte)

Les contraintes de performance : ce sont les contraintes de mémoire (principale ou disque), de temps de réponse, etc.

f) Sous ensemble et priorités d'implémentation : définition des versions correspondantes aux sous ensembles de spécification et détermination d'un ordre de réalisation.

g) Informations de maintenance : détermination des parties susceptibles d'évoluer une fois le logiciel est opérationnel (facilite l'évolution du logiciel).

h) Glossaire : définition des termes techniques utilisés (supposer aucune connaissance technique de la part du client)

i) Index : permettre une utilisation facile du cahier de charges

4.2 Spécification de besoins

Le but de la spécification est de décrire avec rigueur :
- Les données du système (vue statique)
- Les fonctions du système (vue fonctionnelle)
- Les changements d'état et le contrôle du système (vue comportementale)

Trois axes de classification sont à distinguer :

4.2.1 Degré de formalisme
- Spécification informelle : exemple, en langage naturel.
- Spécification semi formelle : notation graphique dont la sémantique n'est pas définie de manière formelle, exemple, UML.
- Spécification formelle : exemple, les Rdp.

4.2.2 Style des énoncés
- Spécification fonctionnelle : description du quoi ➔ Rdp, dfd
- Spécification descriptive : description des propriétés désirées ➔ Modèle E-A

4.2.3 Nature des aspects décrits
- Spécification statique : on décrit ce qui ne change pas dans le système (données) ➔ modèle E-A.
- Spécification dynamique : on décrit ce qui change dans le système (états) ➔ les Rpd

5. Conclusion
- Définition des services attendus par l'utilisateur.
- Analyse des fonctions et de leurs contraintes.
- Importante collaboration de l'utilisateur.
- Utilisation des modèles, méthodes et de techniques statiques ou dynamiques
- Définir les limites (ce que le logiciel doit effectivement réaliser).
- Pas de solutions techniques à ce niveau (complètement détaché des problèmes d'implémentation)
- Envisager des évolutions (prévoir certaines évolutions du logiciel).

La spécification fonctionnelle d'un logiciel

Le but de ce chapitre est de fournir aux étudiants des méthodes de spécification des logiciels en prenant en considération l'aspect statique et dynamique.
Le chapitre est organisé comme suit:

1. **Introduction**
2. **La spécification statique du logiciel : l'analyse structurée**
3. **La spécification dynamique du logiciel : l'analyse structurée temps réel**

1. Introduction

- Spécifier c'est dire quoi mais pas comment.
- Passage de l'expression des besoins à la conception.
- Une étape du cycle de vie d'un logiciel / système la plus créatrice et subjective.
- Une étape où deux aspects sont à considérer: statique et dynamique.
- Les documents de spécifications des logiciels :
  - définitions abstraites des logiciels.
  - destinés aux développeurs.

2. La spécification statique du logiciel : L'analyse structurée S.A (Structured Analysis)

2.1. Présentation générale de la méthode S.A

- S.A est définie par E. YOURDON et T. DEMARCO en 1978-1979.
- Une méthode descendante par affinage successif des traitements ou process (ou encore processus)
  - ➔ aboutir à des traitements décrits en termes de flots de données logiques.
- Une méthode composée d'une suite de diagrammes ordonnée hiérarchiquement
  - ➔ les derniers diagrammes représentent les fonctions élémentaires appelées primitives fonctionnelles ou process primitives.

2.2. Les concepts de base de la méthode S.A

Le langage S.A est composé d'outils graphiques et/ou textuels.

- Les outils graphiques: Diagrammes de transformation de données ou diagrammes de flots de données, DFD (Data Flow Diagrams)
- L'outil textuel: Dictionnaire de données, DD (Data Dictionnary)
- Les outils graphiques et/ou textuels:
  - Diagramme de structure de données, DSD (Data Structure Diagrams).
  - Spécification de process PSPEC (Process SPECification).

2.2.1 L'outil graphique

Le DFD est un réseau d'interconnexion de process (traitements) reliés par les données qui circulent. Le DFD comprend quatre éléments graphiques:

- Le traitement ou process : représenté par un cercle ou une bulle, c'est une unité de travail qui réalise la transformation de la donnée d'entrée en donnée de sortie.

Que signifie une transformation?

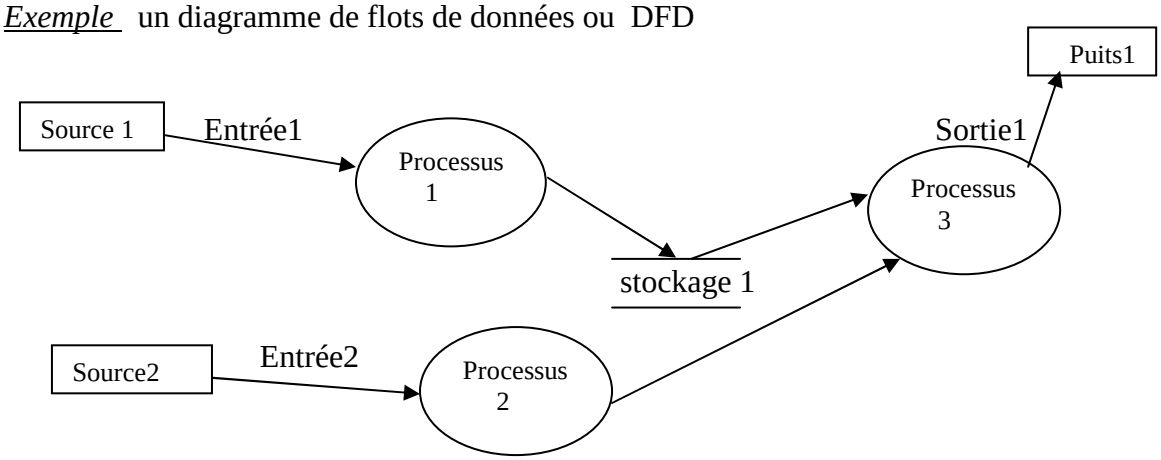
- une transformation de la structure d'une donnée.
- une transformation d'une donnée en une autre.
- un test et en fonction duquel la donnée est aiguillée.

- Le flot de donnée : représenté par un vecteur nommé reliant deux process sur lequel circule un ensemble de données de même nature.

- L'unité de stockage ou le réservoir : représenté par deux traits parallèles et correspond à une zone de rangement des données.
  A ce niveau de représentation de données, il n'y a pas de clés d'accés ni de méthodes de rangement ou de consultation des données.

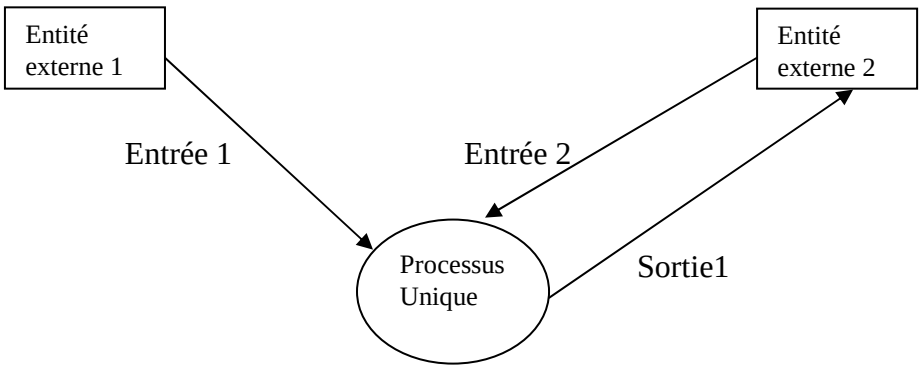
- L'entité externe ou le terminateur : représenté par un rectangle, il indique la provenance (la source) ou la destination (le puits) des données traitées par le système.

Exemple un diagramme de flots de données ou DFD



Mais, un diagramme de contexte de données comprend qu'un seul processus définissant la fonction globale. A ce niveau, il n'y a pas d'unité de stockage

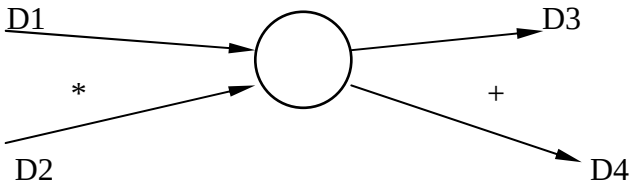
Exemple : un diagramme de contexte de données



Remarques

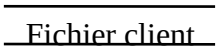
- Chaque processus est identifié par un verbe et un n° correspondant à son niveau hiérarchique.
- Tous les flots de données doivent être nommés.
  L'une des extrémités est reliée au process l'autre au process / un terminateur / une unité de stockage.
  Deux flots de données ne peuvent pas avoir le même nom.

Annotations procédurales d'un DFD

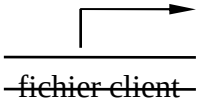


Le caractère * signifie ET, il traduit la conjonction des paquets d'informations.
➔ les deux doivent être présents pour que le process soit exécuté.
Le caractère + signifie OU, il exprime le fait que l'un ou l'autre des paquet est produit par le process.

- Les unités de stockage

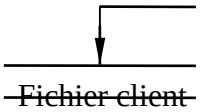


nom descriptif



lecture du fichier

écriture dans un fichier

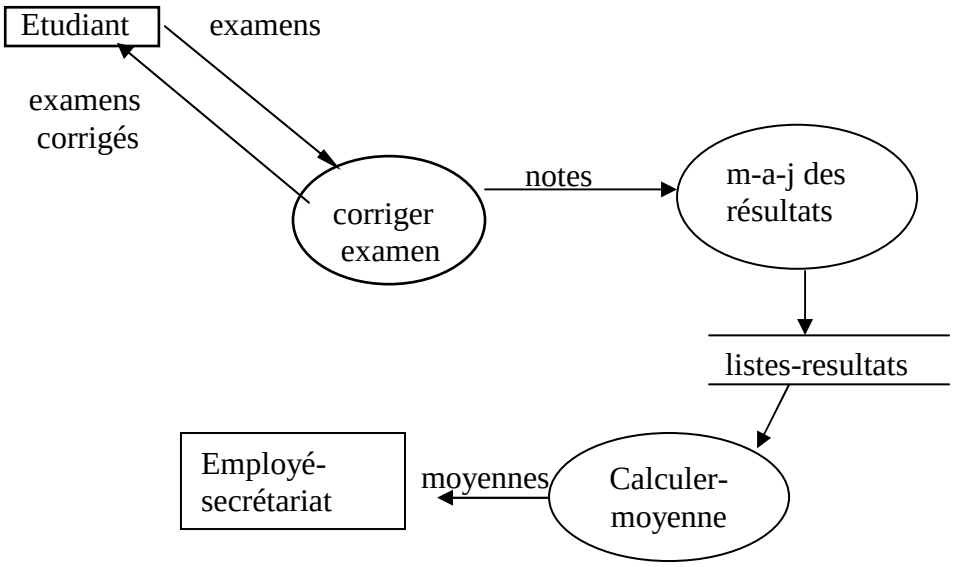


Les opérations sont déduites du sens des flèches

- Les terminateurs

Ils représentent une personne ou un périphérique.
Les terminateurs sont nommés.
Il n'y a pas de méthodologies concernant les flots de données entrant ou sortant des terminateurs.

Exemple : le calcul d'une moyenne



2.2.2 L'outil textuel : le dictionnaire de données

Il est crée et interprété en même temps que les diagrammes de flots de données. Il regroupe la sémantique et la structure de toute donnée apparaissant dans les DFDs.
Il représente un document sur la totalité du système.

Les avantages du dictionnaire de données

- Source de documentation et de référence.
- Amélioration de la communication entre les développeurs
- Réutilisation dans d'autres applications.
- Premier pas vers la création d'une base de données.

Les opérateurs utilisés pour définir la composition d'un élément donné du dictionnaire de données

Symboles	Signification
=	est composé de
+	Séquence; et; avec
max { } min	Répétition; suite de (min----max)
[../../...]	sélection
( )	Optionnel
" "	Chaîne littérale
* *	commentaire

Exemples :

Adresse postale est nom
Adresse de rue
Nom de ville
Code postal

Adresse postale = nom + adresse de rue + nom de ville + code postale

Fichier étudiant est une suite d'enregistrements d'étudiants
Fichier étudiant = {enregistrements d'étudiants}$_1^{150}$

Transaction client est un retrait ou un dépôt
Transaction = [retrait / dépôt]

2.2.3 Les outils graphiques et / ou textuels

L'accent est surtout mis sur la spécification des traitements ou process.
Les primitives fonctionnelles peuvent être exprimées à l'aide de 5 types outils:

- Les algorithmes abstraits
- Les arbres de décision
- Les tables de décision
- Les diagrammes de M.Jackson
- Les diagrammes de N.Schneidermann

Les algorithmes abstraits

- L'idée est de minimiser les ambiguïtés du langage naturel
- C'est un langage structuré limité à 3 types de structures à savoir la structure séquentielle, alternative et répétitive.
- C'est un pseudo code.

a) la séquence ou traitement

```
traitement1
traitement2
traitement3
```

b) Les alternatives simples

```
si <condition > vraie alors
    Traitement
finsi
```

```
si < condition > fausse alors
    traitement
finsi
```

Les alternatives composées

```
si < condition > vraie alors
    traitement1
sinon
    traitement2
finsi
```

```
décider entre
  cas-1 vrai alors traitement1
  cas-2 vrai alors traitement2
  cas-3 vrai alors traitement3
autrement
   Erreur
fin de décider
```

c) Les structures répétitives

(Sortie en tête)

```
Tq <condition > vraie faire
   Traitement
Fin de tq
```

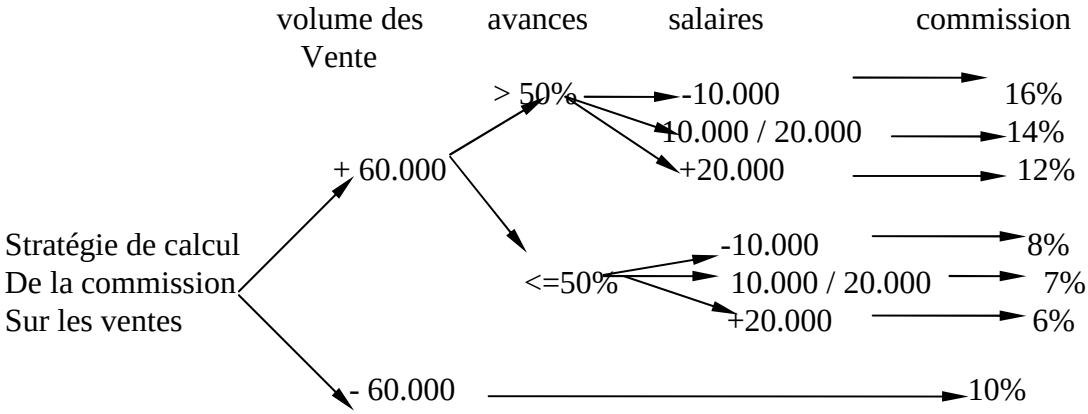
(Sortie en queue)

```
répéter
    traitement
jusqu'à < condition> vraie
```

Les arbres de décision

- C’est une représentation alternative des tables de décision
- Capture les mêmes informations mais sous forme graphique que tabulaire
- Présente une meilleure lisibilité qu’une table de décision
- Prend plus de place en raison de l’espace nécessaire pour représenter tous les branchements.
- Nécessite une évaluation de l’ordre des conditions pour obtenir une simplification du graphique

Exemple



Les tables de décision

- Nécessaires pour décrire des décisions complexes.
- indispensables lorsqu'il y a un grand nombre de combinaisons de conditions.

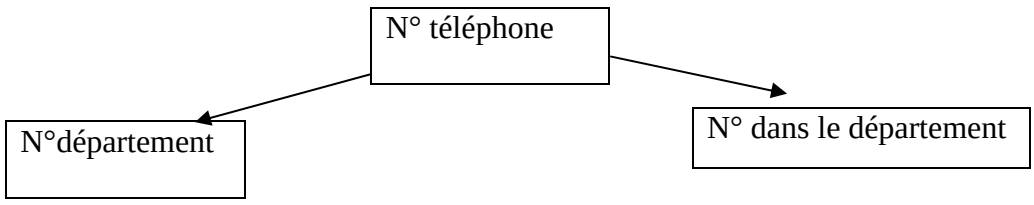
Règles

	1	2	3	4	5
Condition1	F	F	V	V	V
Condition2	V	F	V	F	F
Condition3	F	V	V	F	V

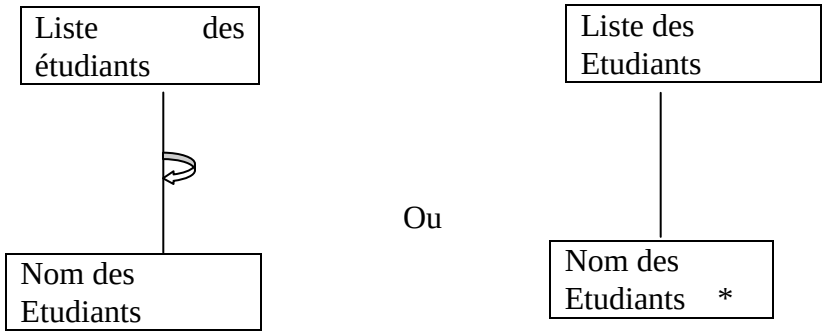
Action1	X				
Action2		X	X		
Action3				X	X
Action4	X	X		X	X

Les diagrammes de M. Jackson

- Séquence

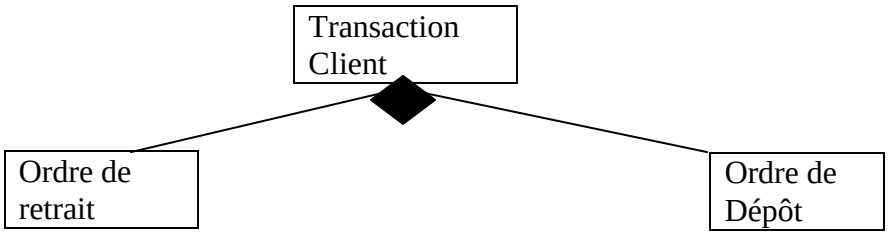


- Répétition



Liste des étudiants = {nom des étudiants}*

- Sélection



Transaction client = {ordre de dépôt / ordre de retrait}

Les diagrammes de N. Schneiderman

- Séquence ou traitement

Traitement1
Traitement2
Traitement3

- Alternatives composées

si

Vrai	Faux
Traitement1	Traitement2

Décider entre

Cas1	Cas2	Cas3	Cas4
Traitement1	Traitement2	Traitement3	Traitement4

- Répétitives

(sortie en tête)

Tq condition	Vraie
	Traitement 1 Traitement2

(sortie en queue)

	Traitement1 Traitement2
Jusqu'à condition	Vraie

2.3 Conclusion

Les similitudes avec le sous-ensemble IDEF0 de la méthode SADT sont :

- Flots de données dans SA = flèches de données entre les activités dans SADT
- Les diagrammes sont ordonnés hiérarchiquement :
  A0, A1, ….. ≡ DFD0, DFD1, …….
- Le raffinage de chacun des process en SA ≡ des boites en IDEF0
- Le dictionnaire de données en SA ≡ Le glossaire en IDEF0.

Les différences sont les suivantes:

- Les diagrammes d'activité IDEF0 sont plus riches qu'un DFD en SA (flèches de contrôle et les mécanismes).
- Au niveau plus bas de la spécification en SA, l'étude se poursuit par la description fonctionnelle de tous les process primitifs.
- La syntaxe graphique d'un DFD est plus libre qu'un actigramme.
- La méthode SA ➔ phase de spécification fonctionnelle d'un logiciel
  La méthode IDEF0 ➔ phase de conception d'un système.
- SA est destiné pour le développement des logiciels industriels.

3. La spécification dynamique du logiciel : l'analyse structurée temps réel SA-RT (Structured Analysis - Time Real)

3.1 Présentation générale de la méthode SA-RT

- L'extension temps réel a été effectuée par P.Ward et S. Mellor en 1984-1985 d'une part et par I.Pirbhai et D. Hatley en 1986-1987 d'autre part.
- SA-RT est considérée comme une méthode la plus utilisée dans le monde.
- SA-RT est supportée par plusieurs AGLs et a été choisie par d'importantes sociétés (Boeing, Aircraft Compagny etc..).

3.2 Concepts de base de la méthode SA-RT

- Pourquoi SA-RT?
  - l'analyse structurée SA est insuffisante pour modéliser l'activation des process.
  - L'analyse structurée SA s'intéresse à la vue statique des fonctions que le logiciel devra assumer.
- SA-RT sert à implémenter le logiciel qui sera implémenté dans des systèmes temps réels:
  - les systèmes combinatoires où les signaux de sortie sont complètement définis par les valeurs des signaux d'entrée.
  - Les systèmes séquentiels où les signaux de sorties sont déterminés par les signaux d'entrée et les états internes.
- Pour spécifier un logiciel, SA-RT dispose d'un ensemble
  - d'outils graphiques et/ou textuels issus de SA.
  - d'extensions graphiques et/ou textuels pour le temps réel.

3.2.1 Les outils graphiques et textuels issus de la méthode SA

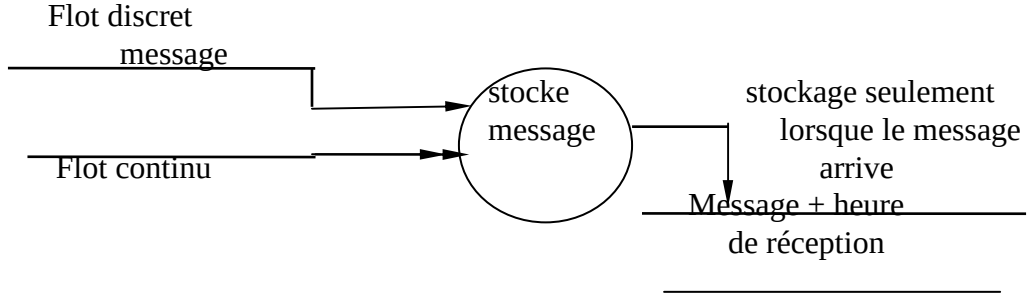
- L'outil graphique : Le diagramme des flots de données DFD: il représente l'ensemble des traitements qui manipule l'information et se compose de 4 éléments de base:
  - le processus de données, représenté par
  - le flot de données, représenté

en notation de Ward- Mellor par

 qui représente les flots de données discrètes et qui véhiculent une donnée valide à des instants précis.

 qui représente les flots de données continus et qui véhiculent une donnée continuellement valide dans le temps.

Exemple



en notation Pirbhai - Hatley

pas de distinction entre flot discret et flot continu.

----------▶ qui montre qu'une donnée arrive à des instants précis.

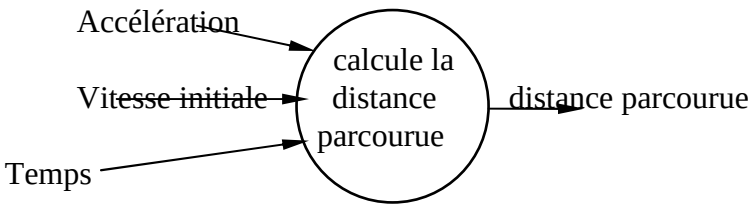
- l'unité de stockage de données ou réservoir : représentée par ═══ quelque soit l'école.
- l'unité de stockage des évènements : représentée par ---------- selon Pirbhai-Hatley
- L'unité externe ou le terminateur : représenté par ▭ quelque soit l'école.

- L'outil textuel : le dictionnaire de données (DD) sert à :
  La définition de chaque flot de données et de chaque stockage utilisée dans le DFD.

- Les deux outils textuels et / ou graphiques
  - le diagramme de structure de données (DSD) pour la représentation de structure de données complexes.
  - les spécifications de process ou mini-spécification fonctionnelle

  Pour l'analyse structurée, les derniers niveaux de raffinage décrivent des traitements élémentaires appelés primitives fonctionnelles permettant de définir la transformation de flots de données entrant en flots de données sortant. En plus des outils de spécification des process décrit dans la méthode SA, on dispose
  - Des équations (process mathématique),
  - Des fonctions propres à la gestion de bases de données (PROJECT, JOIN, SELECT),
  - Ainsi que d'autres commentaires.

Exemple



$$\text{Distance parcourue} = \frac{\text{accélération} * (\text{temps})^2}{2} + \text{vitesse initiale} * \text{temps}$$

3.2.2 Les extensions graphiques et textuelles pour SA-RT proposées

Cette extension comprend:

- L'outil graphique: le diagramme de flots de contrôle CFD (Control Flow Diagrams)
  Il est couplé au DFD de même niveau et il repose sur 4 éléments:
  - Flots de contrôle ou flots d'évènements ordinaires appelés signaux
  - Flots de contrôle « prompts »
  - Les process de contrôle
  - Les stockages de contrôle ou d'évènements

- Les flots de contrôle ou signaux

Les évènements représentés par ----------► selon les deux écoles et ils expriment une occurrence dans le temps.
Les signaux ne véhiculent pas de contenu pour Ward-Mellor contrairement à Hatley-Pirbhai.

- les données discrètes :

propagent une valeur uniquement valide à des instants précis et disparaissent dès que les données ont été consommées. Elles sont représentées par:

——► selon Ward-Mellor
----► selon Hatley- Pirbhai

Les flots de contrôle « prompts »
Uniquement formalisés par Ward-Mellor et représentés par ----------► et portent l'une des mentions suivantes:

- E (Enable) : flots de contrôle d'activation
- D (Disable) : flots de contrôle de désactivation
- T (Trigger) : flots de déclenchement

Ils sont générés par les process de contrôle et sont connectés aux process de données qu'ils activent, désactivent ou déclenchent

- Les process de contrôle

Ce sont des éléments fondamentaux de la méthode SA-RT et ils sont représentés par :

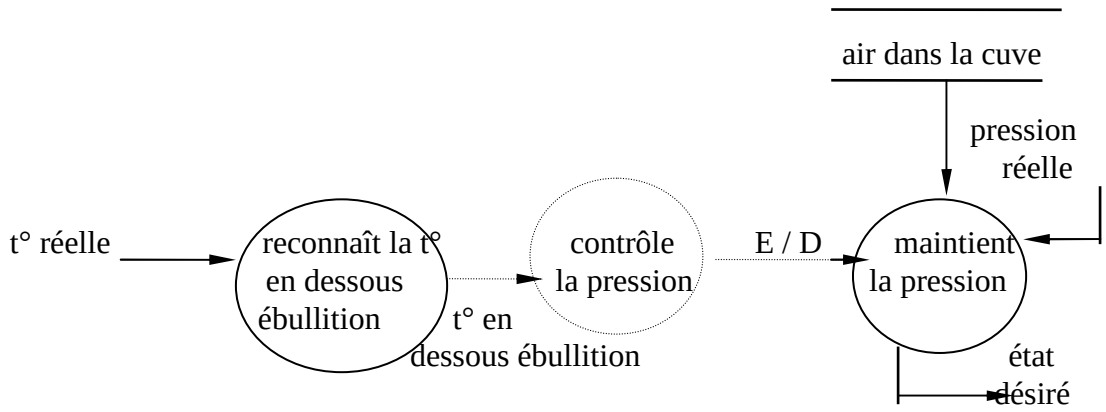
◯ selon Ward-Mellor

| selon Pirbhai-Hatley

- Les stockages des flots de contrôle ou stockage des flots d'évènements

Ils sont proposés uniquement par Pirbhai-Hatley et Ils mémorisent le nombre d'occurrence de flots de contrôle ou d'évènements.
Ils sont représentés par ----------

Exemple



- L'outil textuel : le dictionnaire de données (DD) sert à la
  - définition de chaque flot de contrôle
  - définition de chaque stockage de flot de contrôle ou d'évènements selon Ward-Mellor

- Les deux outils graphiques et textuels

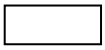
1. la spécification de contrôle CSPEC (Control SPECification) :
   La description la plus complète possible de la logique de fonctionnement du process.

   - En notation de Ward-Mellor:
   Un diagramme état-transition STD (State Transition Diagram) ou
   une table état-transition STT (State Transition Table)

   - En notation de Pirbhai-Hatley :
   Un diagramme état-transition STD ou une table état-transition STT ou
   une matrice état-évènement SEM (State Event Matrice).
   Une table d'activation de process PAT (Process Activation Table).
   Une table de décision.

a) La spécification de contrôle selon Ward-Mellor

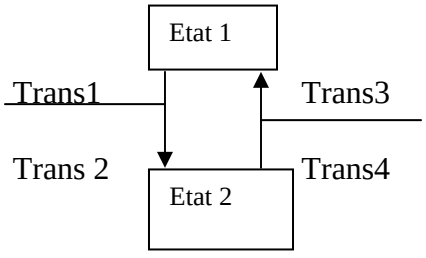
a1) Diagramme état-transition STD (State Transition Diagram) : met en évidence les différents fonctionnements du process de contrôle. Il se compose :

- d'états représentés par 

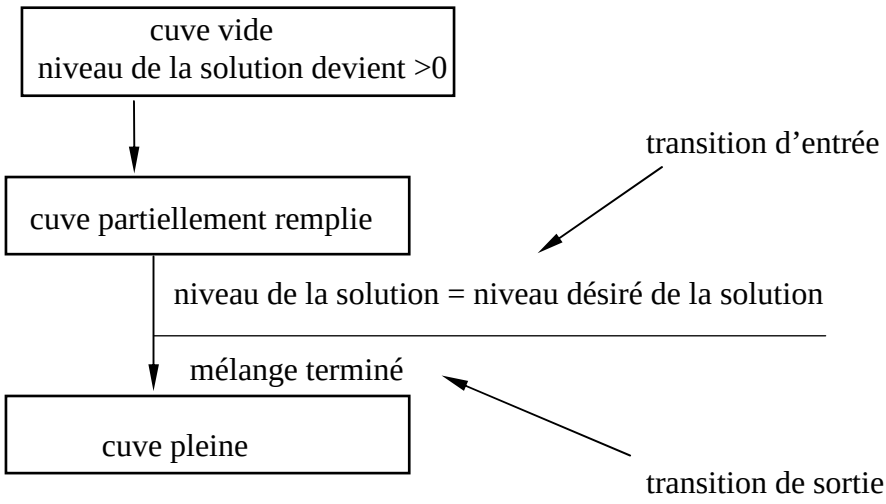
- de transitions qui indiquent le changement du mode de fonctionnement de l'objet analysé et ils sont représentés par de l'état précédent à l'état suivant.

Deux types de transitions sont à distinguer :
- transition d'entrée : qui indique le signal provoquant le changement d'état.
- transition de sortie : qui indique les signaux provoquant un changement de mode de comportement à un autre endroit, décrit par un autre diagramme STD, ainsi que la liste des process à activer ou à désactiver.

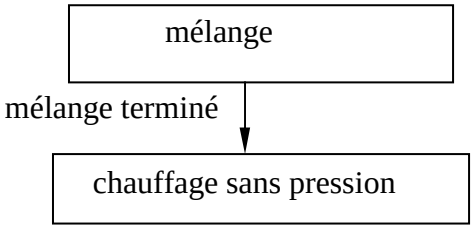
Exemple 1



Exemple 2



La transition de sortie provoque un changement de mode dans un autre diagramme STD.



Remarques

- L'état initial est l'état actif avant la reconnaissance de n'importe quelle transition d'entrée.
- A un instant donné, un et un seul état du diagramme STD est actif.
- La transition d'entrée porte le nom de l'évènement qui provoque le changement de mode.
- Plusieurs diagrammes STD peuvent interagir entre eux.

a2) Table état-transition (STT)

Un diagramme état-transition peut être représenté à l'aide d'une table état-transition où:

- les états sont les lignes de la table,
- les transitions d'entrée sont les colonnes de la table et
- le nouvel état et la transition de sortie représentent l'intersection de la ligne et de la colonne.

Transition / État	Trans1	Trans3
Etat 1	état suivant: Etat 2 Transition de sortie: Trans2	état suivant : aucun Transition de sortie : aucune
Etat 2	état suivant : aucun Transition de sortie : aucune	état suivant: Etat 1 Transition de sortie: Trans4

b) la spécification de contrôle selon Pirbhai-Hatley

b1) Le diagramme état-transition (STD) est identique à celui de Ward-Mellor auquel une table d'activation de process est rajoutée permettant d'évoquer explicitement les process activés, désactivés ou déclenchés suite à des flots de contrôle ordinaires ou prompts.

Tables état-transition STT avec le même principe de celle de Ward-Mellor

état courant	évènement	actions	état suivant
EC1	E1	A1	EC2
EC2	E2	A2	EC3
EC3	E3	A3	EC1

b2) Table d'activation des processus (PAT)
Elle consiste à définir l'action permettant d'activer ou de désactiver un ou plusieurs process. En entrée, on a l'action à exécuter et en sortie, le nom du (des) process à activer ou à désactiver de façon séquentielle ou concurrente.

Process / Action	process1	process2	process3
A1	1	0	0
A2	0	1	2
A3	1	2	2

Pour cette table d'activation, par exemple, pour l'action A1, il faut activer le process1 et désactiver le process2 et le process3. Pour l'action A3, il d'abord activer le process1 puis les process2 et process3 en parallèle.

b3) Matrice état-évènement (SEM)
Elle permet de montrer les différentes actions à exécuter par le système.

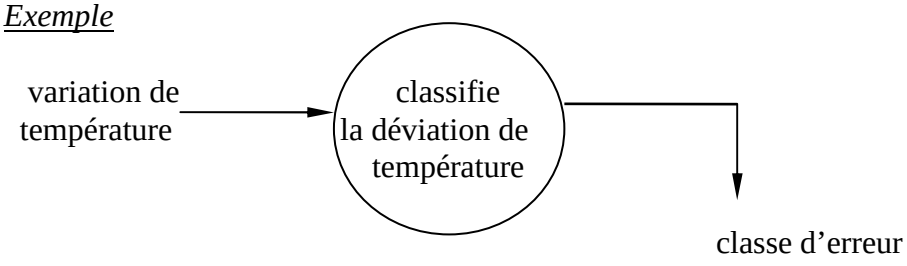
Évènement / État	E1	E2	E3
EC1	A1/ EC2	A2 / EC3	
EC2		A3 / EC1	A4 / EC3
EC3		A3 / EC2	NA

Le label NA signifie que l'évènement ne peut pas être reconnu. Une case vide signifie que l'évènement ne peut pas causer de transition.

b4) Table de décision

Elle est nécessaire lorsqu'il y a un grand nombre de combinaisons de conditions mais aussi pour une explication simple de certaines transformations.

Exemple



valeur de la variation de la température	classe d'erreur
<2 2 < t°< 5 5 < t° < 8	normal warning urgence

2 La spécification du temps

La table des temps de réponse représente un élément indispensable pour l'analyse d'un logiciel d'une application temps réel. Elle permet de préciser les informations suivantes :

- la fréquence de répétition des signaux,
- le temps de réponse entre le signal d'entrée et le signal de sortie correspondant.

Exemple :

Entrée	événement	sortie	événement	Temps de réponse
Demande de vitesse	entrée	Vitesse	affiché	0.5 sec max

3.3 Conclusion

- La méthode SA-RT sert à la modélisation d'un logiciel à fortes contraintes de temps.
- Elle présente un certain flou dans la syntaxe de la méthode concernant les règles de circulation des données entre process qui ne sont pas clairement définies
- Le DFD, le DD et la PSPEC résultent d'une interprétation humaine.
- Elle est partiellement supportée par les AGLs (vérification des données).
- Certains contrôles automatiques sont difficiles à envisager (probablement) actuellement.

La conception d'un logiciel

Le but de ce chapitre est de fournir aux étudiants des méthodes de conception des logiciels notamment la conception structurée et la conception orientée objet
Le chapitre est organisé comme suit:

1. Introduction
2. Types de conception
3. Définition d'une architecture
4. Caractéristiques d'une bonne conception
5. Stratégies d'une conception
6. Concevoir pour changer
7. Qualités d'un bon design
8. Les différents types de module
9. Autres heuristiques de la composition et décomposition modulaire
10. La conception structurée
11. La conception orientée objet

1. Introduction

- C'est un pont entre la définition des besoins et l'implémentation ; c'est une activité itérative / incrémentale qui transforme progressivement les besoins vers un produit final.
- La conception s'intéresse au comment (la solution) et non au quoi (le problème)
- C'est un processus créatif qui demande du savoir-faire ; Elle ne s'apprend pas dans un livre mais par la pratique et l'étude des systèmes existants ainsi que par l'expérience.
- Une bonne conception aide au développement d'un logiciel efficace, fiable et de qualité.
- Les documents élaborés en phase de conception préliminaire sont : dossier de conception préliminaire et le plan d'intégration.

2. Types de conception

- Conception architecturale :
  - Structure et organisation générale du logiciel à concevoir.
  - La première étape qui consiste à définir la fonction de chaque composant du logiciel et leurs relations fonctionnelles.
- Conception détaillée :
  - Etape qui consiste à détailler les résultats de l'analyse fonctionnelle jusqu'à un niveau suffisant pour en permettre le codage dans un langage de programmation choisi.
  - Concevoir pour chaque composant : algorithmes / structures de données / interfaces.

3. Définition d'une architecture

Une architecture : est une structuration du logiciel à développer en un ensemble de composants ou modules

- ayant entre eux des relations bien définies
- possédant certaines propriétés requises
- et respectant certaines contraintes.

Qu'est-ce qu'un module?
Une partie d'un logiciel correspondant à une unité fonctionnelle reliée aux données qu'il manipule.
Une unité fonctionnelle : une fonction élémentaire / une fonction complexe (ensemble de fonctions).

Exemples

Pascal	module = procédure
C	module = fonction
Fortran	module = sous-programme
Ada	module = package

Pourquoi une structure modulaire ?

- Structure maîtrisable : « diviser pour régner »
- Qualités acquises d'un logiciel : facilité de modification,
  facilité de vérification,
  facilité de réutilisation.
- Changement facile : caractéristique principale des logiciels.

Relation : inclure / utilise / call
Propriété : validité / sécurité
Contraintes : localisation / matériel / taille des modules

D'autres définitions au terme architecture logicielle ont été proposées dont:

- "A Software architecture is the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time" (Garlan et Perry, IEEE 1995)
- "A Software architecture is an abstract system specification consisting primarily of functional components described in terms of their behaviors and interfaces and component-component interconnections" (Hayes-Roth,1994)
- "The software architecture of a program or a computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them" (Bass, Clement and Kazman, 1997)

4. Caractéristiques d'une bonne conception

Une bonne conception en modules devra favoriser :

- Une forte cohésion : les éléments ne sont pas réunis par hasard dans un même module. Ils forment un tout dans le but de réaliser une tâche commune.
- Un faible couplage : les modules sont relativement indépendants.
- L'abstraction : prendre en considération ce dont on a besoin pour l'utilisation souhaitée (considérer uniquement les aspects importants d'un système à un moment donné en faisant abstraction des autres aspects.
- Le masquage d'information : seules les informations qui servent à la communication avec d'autres modules sont visibles de l'extérieur du module.

5. Stratégies de conception

Comment procéder à la décomposition en modules ?

- Stratégie descendante :

Avantages :

- Vue d'ensemble du problème et de ce qu'on désire réaliser
- Facilite la compréhension des problèmes et de leur décomposition « diviser pour régner »

Inconvénients :

- Les sous problèmes ont tendance à être analysés de façon isolée.
- Ne favorise pas la réutilisation.
- Principe de généralisation et de masquage d'information non mis à profit.

- Stratégie ascendante :

Avantages :

- Permet d'identifier ce qu'il y a de commun entre les modules et permet d'appliquer le principe de masquage d'information et de généralisation.
- Favorise la réutilisation

Inconvénients :

- Pas de vue d'ensemble.
- Risque de consacrer beaucoup d'efforts à réaliser un module qui ne sera pas utilisé.

- Stratégie mixte :

Un bon concepteur appliquera une stratégie mixte.

↓ - On commencera par une stratégie descendante pour identifier les sous systèmes cibles.

↑ - On réalisera ensuite la synthèse des sous systèmes en termes de hiérarchie de modules réutilisables construits en appliquant les principes de généralisation et de masquage d'information.

6. **Concevoir pour changer**

Un des principaux objectifs de l'activité de conception : développer un design qui facilite le changement. Deux sources de vrais changements :

Changements (perfectionnements) du logiciel imposés par de nouveaux besoins des clients ou utilisateurs ➔ maintenance perfective.

Changements (adaptations) imposés par la modification de l'environnement matériel / social ➔ maintenance adaptative.

6.1 types de changements

- Changement d'algorithmes : améliorer l'efficacité pour mieux répondre aux contraintes de sécurité, réduire l'espace mémoire.

  Exemple

  Tri par bulle ➔ tri rapide

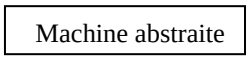
- Changement de représentation de données : améliorer l'efficacité, réduire l'espace utilisé, offrir plus de flexibilité.

  Exemple

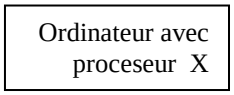
  Tableau ➔ liste chaînée

- Changement au niveau de la machine abstraite sous jacente
  Machine abstraite : logiciel installé sur un ordinateur permettant de simuler le fonctionnement d'un dispositif matériel qui ne pourrait pas fonctionner avec cet ordinateur.

Programme SQL



SGBD



Processeur Pentium

C'est un changement apporté à l'environnement d'exécution, pas au système développé, pour réduire l'espace mémoire, optimiser les calculs.

➔ adaptation nécessaire, par exemple reformater la BD pour satisfaire les nouvelles contraintes de stockage des données par le SGBD.

- Changement au niveau des périphériques

Ces des changements apportés aux périphériques pour les rendre plus autonomes, plus intelligents.

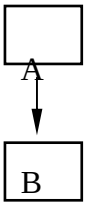
➔ adaptation nécessaire pour s'accommoder de ces modifications aux périphériques.

7. Qualités d'un bon design

Un bon design devrait favoriser l'indépendance des modules. Pour l'évaluer, on se base sur les concepts suivants : le couplage et la cohésion.

7.1 Le couplage

- Le couplage est la mesure d'interdépendance d'un constituant par rapport à un autre.
- Le couplage définit le mode de communication interconstituant



A appelle B
B remplit sa fonction
B redonne la main à A dès qu'il remplit son rôle

7.2 types de couplage

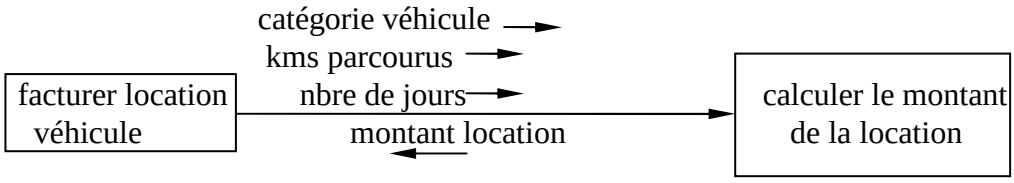
couplage

couplage dangereux et difficile à comprendre	- par contenu - de données communes - de contrôle	**couplage fort**
couplage traditionnel	- de collection **- de données** - absence de couplage	**couplage faible**

7.2.1 Le couplage de données

- correspond à la transmission de données nécessaires au fonctionnement de modules en communication.

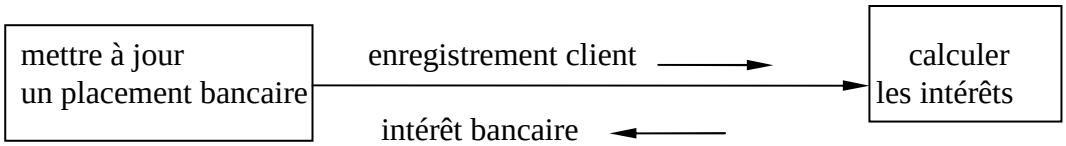
Exemple



7.2.2 Le couplage de collection

- Correspond à deux modules qui font référence à une même structure de données

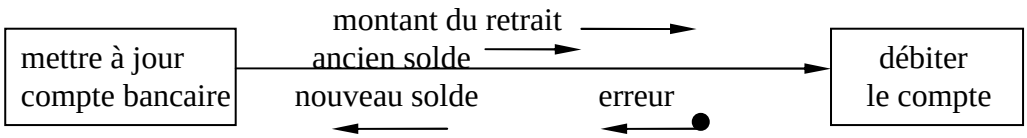
Exemple



7.2.3 Le couplage de contrôle

- Correspond à un module qui transmet à un autre une information destinée à contrôler sa logique interne.

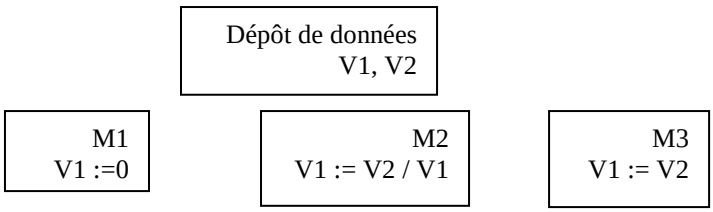
Exemple



7.2.4. Le couplage de données communes

- Correspond à deux modules qui font référence à un même bloc de données globales.

Exemple



M1 risque d'entraîner une erreur dans M2 via la variable V1.

7.2.5 Le couplage par contenu

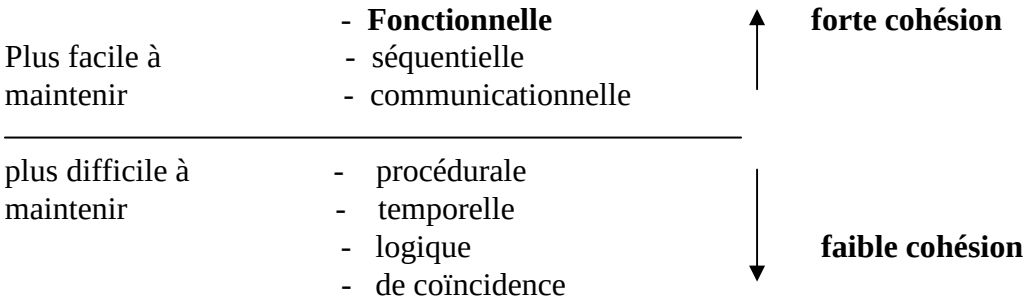
- Survient quand un module :
  - influe sur l'exécution d'une instruction appartenant à un autre module.
  - se réfère ou modifie les données internes d'un autre module.
  - généralement impossible dans les langages de haut niveau, seulement en assembleur.

7.3 La cohésion

- la cohésion est la mesure de la force d'association des éléments fonctionnels d'un constituant.

7.4 Types de cohésion

Cohésion



7.4.1 La cohésion fonctionnelle

- Tous les composants contribuent à remplir une seule et même fonction.

Exemple:

lire un enregistrement
calculer la trajectoire du missile

7.4.2. La cohésion séquentielle

- Ensemble d'actions différentes qui s'enchaînent séquentiellement

Exemple:

lire
calculer
imprimer

7.4.3 La cohésion communicationnelle

- Ensemble d'actions différentes qui se réfèrent aux mêmes paramètres d'entrée/ sortie.

Exemple:

chercher le titre d'un livre
chercher l'auteur d'un livre
chercher le prix d'un livre

7.4.4. La cohésion procédurale

- Exécution de multiples actions n'ayant pas de relation entre elles.

Exemple

écrire
lire
éditer

7.4.5. La cohésion temporelle

- Exécution de multiples actions dont l'unique relation est le temps.

Exemple:

se lever
se laver
faire la prière
déjeuner

7.4.6. La cohésion logique

- Les actions peuvent être considérer comme faisant partie d'un même thème mais indépendantes les unes des autres.

Exemple

Voyager en voiture
Voyager par train
Voyager en avion

7.4.7. La cohésion de coïncidence

- Regroupe des actions n'ayant entre elles aucune relation fonctionnelle (réunies par pure commodité).

Exemple

Préparer la voiture
Faire des gâteaux
Etudier

Remarque

Une bonne conception est atteinte avec **un haut degré de cohésion et un faible couplage**. L'avantage des systèmes à forte cohésion et à faible couplage est qu'il est possible de remplacer un composant quelconque par un composant équivalent avec peu ou pas de changements dans les autres parties du système.

8. Les différents types de modules

Cinq types de modules différents sont à définir compte tenu des concepts de structuration et plus particulièrement des mesures de couplage et de cohésion.

8.1 Les modules de données

- Gèrent un ensemble de données reliés fonctionnellement, indépendamment de leur implémentation ou de leur représentation.

Fonctionnellement, deux types de données peuvent être identifiés:

- Données statiques:
  - Décrivent l'état ou l'environnement ne subissant aucune modification en cours d'exécution.
  - Sont initialisées une et une seule fois.
  - Sont accessibles en lecture uniquement.

- Données dynamiques:
  - Décrivent un état modifié en cours d'exécution.
  - Sont accessible en lecture et en écriture.

8.2 Les modules de traitement

- Gèrent un ou plusieurs processus effectuant une partie des traitements spécifiés ainsi que les tâches activées sur ces processus.

8.3. Les modules fonctionnels

- Regroupent les composants réalisant une même famille de fonctionnalité auxquelles sont associés des processus et des tâches.

8.4. Les modules de communication

- Représentent une interface de communication entre les modules de traitements.
- Prennent en considération les communications internes et externes.

8.5 Les modules couche

- Concernent les logiciels construits autour d'une architecture en couches.
- Peuvent être constitués de modules de données, de modules de communication et de modules fonctionnels.

9. Autres heuristiques de la composition et décomposition modulaire

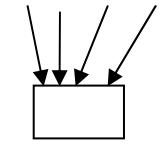
9.1 L'initialisation et la fin de traitement

Ces modules doivent se situer, dans la hiérarchie, aussi proches que possible des fonctions qu'ils initialisent ou qu'ils clôturent

9.2 L'entrance (Fan in)

- Représente le nombre de modules qui l'appellent.

Exemple:

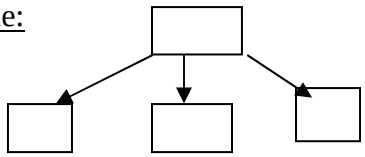


Fan in = 4

9.3 La sortance (Fan out)

- Représente le nombre de modules qu'il appelle.

Exemple:



Fan out = 3

Dans une bonne structure modulaire, le nombre d'appels sortants (**fan out**) doit rester relativement **faible** alors que le nombre de modules entrants (**fan in**) peut être **élevé** ➔ le module est souvent réutilisé.

10. La conception structurée

10.1 Introduction

- La conception structurée SD (Structured Design) proposée par de nombreux auteurs (en 1974 par W. Stelens, G. Meyers, L. Constantine ; en 1976 par C.P. Gane, T.

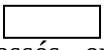
Sarson ; en 1980 par Pages Jones) est très liée à la méthode S.A définie par E. Yourdon et T. Demarco.

- Le passage de S.A (qui modélise le problème) à la conception structurée (qui modélise la solution) est facile.
- SA-SD couvre les deux étapes (analyse et conception) du cycle de vie du logiciel.
- SA : qu'est ce que le système doit faire; SD : comment le système doit être fait.

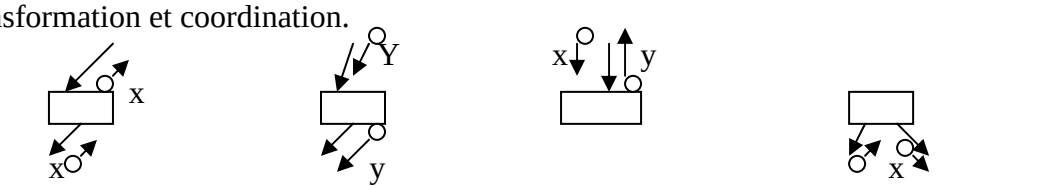
10.2 Concepts de base de la méthode SD

La méthode SD est composée d'outils graphiques et textuels suivants :

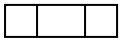
10.2.1 Le diagramme de structure SC (Structured Chart) : Il reflète une structure hiérarchique et fonctionnelle du logiciel et il est construit à partir :

- de modules représentés par 

Les modules peuvent être classés en types entrée (afférent), sortie (efférent), transformation et coordination.



entrée sortie transformation coordination

- de modules prédéfinis représentés par 

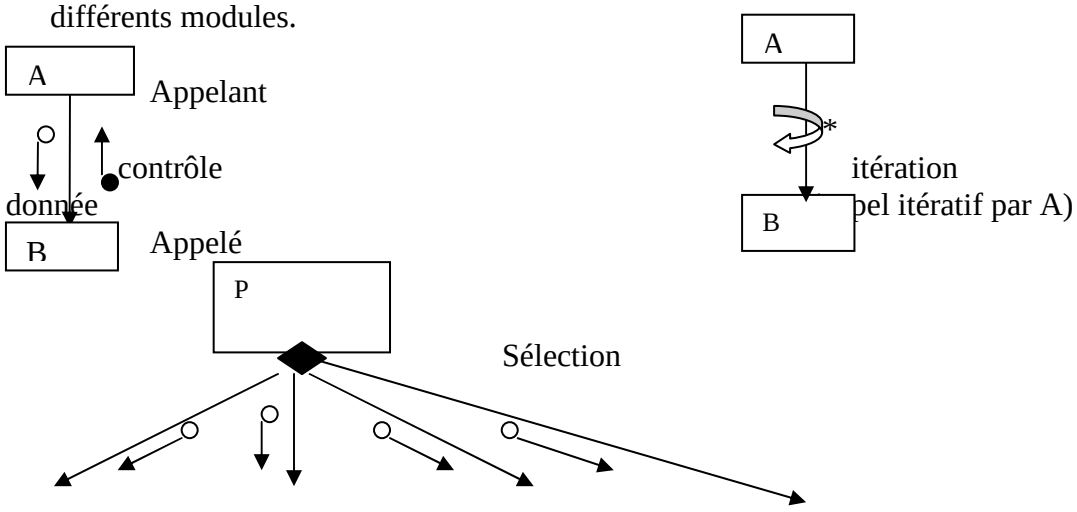
- de flèches de liaison entre les différents modules qui correspondent à la relation d'appel entre modules.

- de petites flèches avec un rond qui indiquent la direction des mouvements de données et de contrôles.

- De plusieurs signes (arc de cercle, losange, etc.) pour préciser l'itération, la sélection, etc.

- Le dictionnaire de données crée lors de l'analyse structurée et complété avec des informations issues de la conception structurée.

10.2.2. La description des modules MD (Module Description) écrit en PDL (Program Description language) indispensable aux programmeurs pour implémenter les différents modules.



10.3 La démarche méthodologique de la conception structurée

- Le passage du problème modélisé par un DFD à la solution modélisée par SC n'est pas directe.
- Il nécessite un ensemble de règles à suivre et de stratégies à essayer.
- Plusieurs méthodes permettent ce passage dont la conception axée sur les transformations (transform centered).

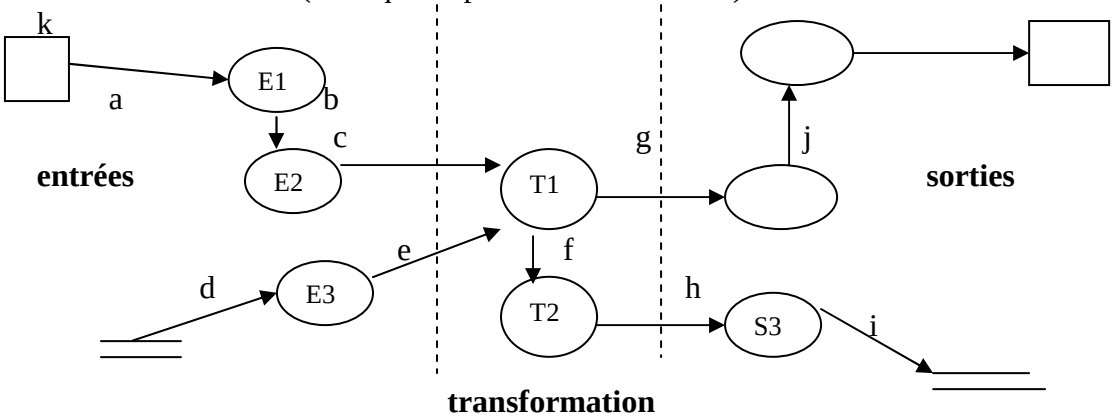
10.3.1 La conception axée sur les transformations

- Le support de départ est un DFD, résultat de l'analyse structurés.
- Le résultat final est un SC, résultat de la conception structurée.
- La procédure à suivre s'appuie sur trois opérations :
    - Localiser le centre de transformation de plus haut niveau.
    - Construire un premier diagramme de structure.
    - Remanier et optimiser le diagramme de structure.

10.4 Applications

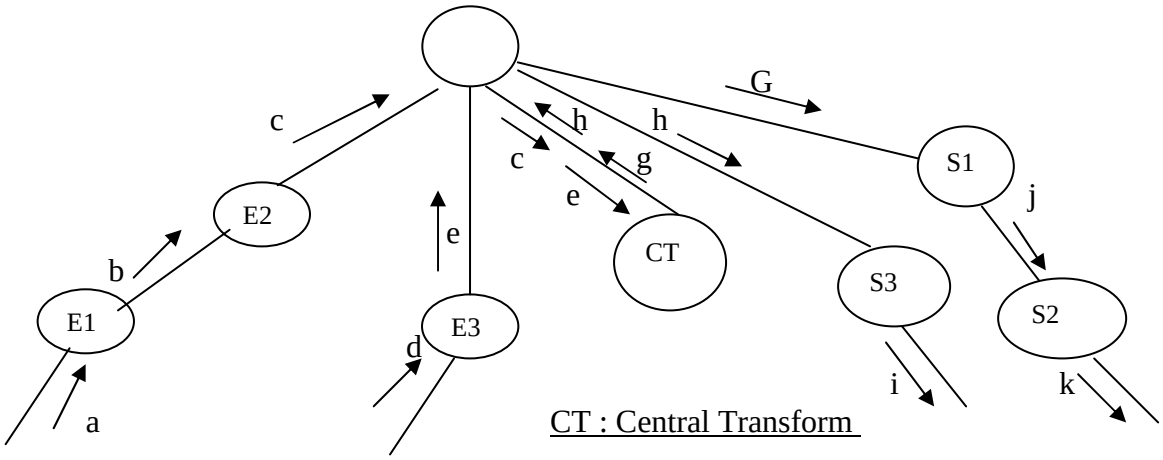
Deux procédés pour aboutir à la construction d'un SC dont la procédure décrite en 10.3.1 (procédure manuelle):

a) localiser le centre de transformation de plus haut niveau
    - parcourir le DFD et localiser le centre de transformation.
    - Détacher les flots afférents (flots qui apportent l'information) et les flots efférents (flots qui emportent l'information)

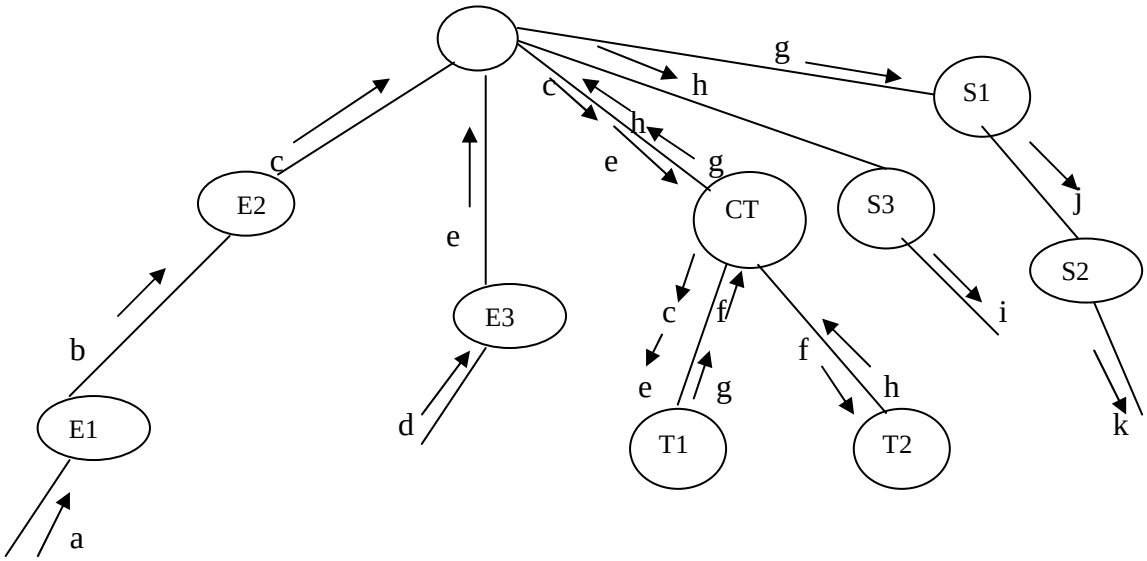


b) Construire un premier diagramme de structure
    - Choisir un bon candidat au poste de chef, l'utiliser comme module de plus haut niveau.

- Dans le cas contraire, désigner un nouveau chef, considérer le centre de transformation comme une boite unique et accrocher le centre de transformation, les afférents et les efférents au nouveau chef.

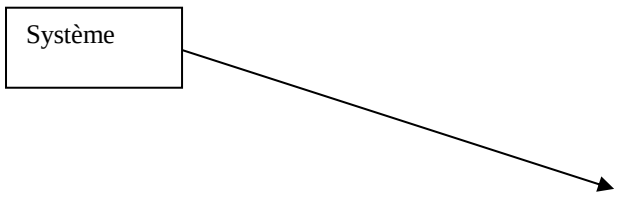


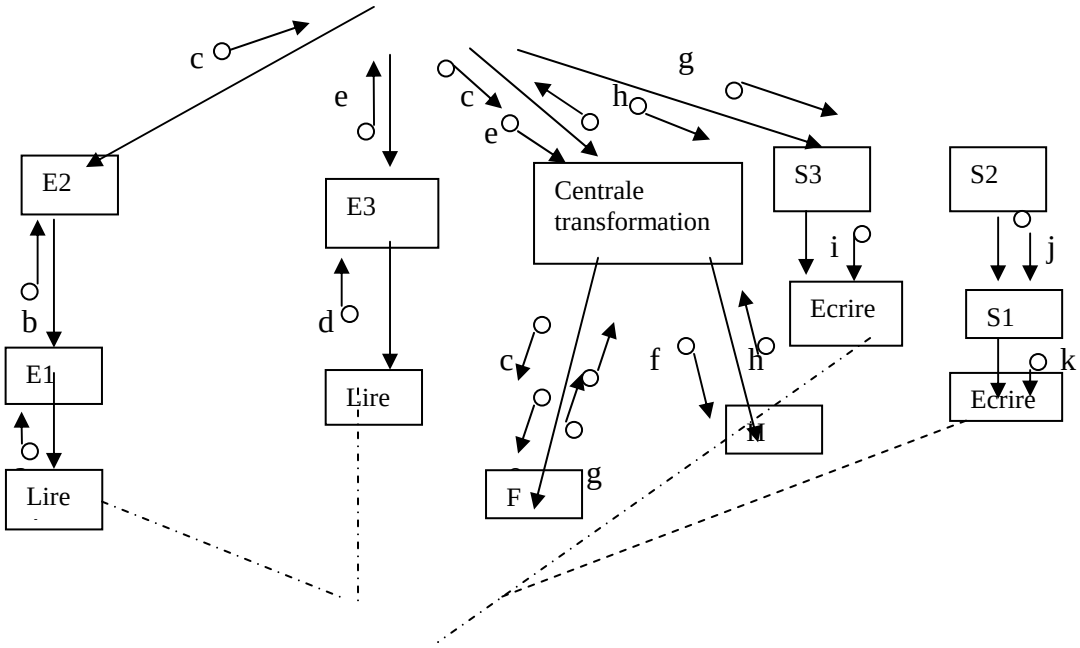
CT : Central Transform



Raffinage de la transformation centrale

c) Remanier et optimiser le diagramme de structure
   - Ajouter les modules de lecture et d'écriture pour accéder aux unités externes (source et puits) et aux unités de stockage.
   - Organiser les modules afférents et efférents.
   - Ajouter les modules de traitements d'erreurs.
   - Ajouter les modules d'initialisation et de terminaison.
   - Choisir les noms des modules illustrant bien leur rôle.
   - Vérifier toutes les données et les contrôles entre les modules.





Modules d'E/S

10.5 Conclusion

Les principaux avantages de la conception structurée SD sont les suivants:
SA-SD couvre les trois phases du cycle de vie du logiciel

- La phase de spécification et la méthode SA.
- La phase de conception préliminaire et le diagramme de structure.
- La phase de conception détaillée et la description des modules MD

SASD présente une pluralité des outils permettant la description des différents aspects du logiciel ; Les outils utilisés sont : DFD, DD, PSPEC, SC, MD, PDL.

Les inconvénients de cette méthode sont :

- Le DFD devient illisible si le nombre de process apparaissant à un niveau donné est important.
- Les flots de données entrant et sortant apparaissent de la même façon dans le diagramme fils.
- Il est difficile de savoir à quel niveau il faut introduire les unités de stockage.
- Le passage de l'analyse structurée à la conception structurée est difficile (il n'existe pas une unique transformation).
- L'importance des flots de données n'est pas prise en compte (quantité d'information importante ou non).
- Seul l'aspect statique est décrit (pas de symbolisme pour préciser les règles de

Les méthodes fonctionnelles

- Ont leur origine dans le développement des langages procéduraux.
- Plus orientées vers les traitements que vers les données.
- Mettent en évidence la ou les fonctions à assurer.
- Proposent une approche hiérarchique, descendante et modulaire en précisant les liens entre les différents modules.
- Utilisent généralement des modèles/outils de type DFD.
- Les méthodes fonctionnelles les plus connues sont : SADT, SA-RT, SA-SD.

11
Ur

- propose une démarche distinguant les étapes du développement dans le cycle de vie du logiciel et exploitant les principes fondamentaux : modularité, abstraction, réduction de complexité.

- propose des formalismes (langages) et des types de documents (modèles) qui facilitent la communication, l'organisation et la vérification.

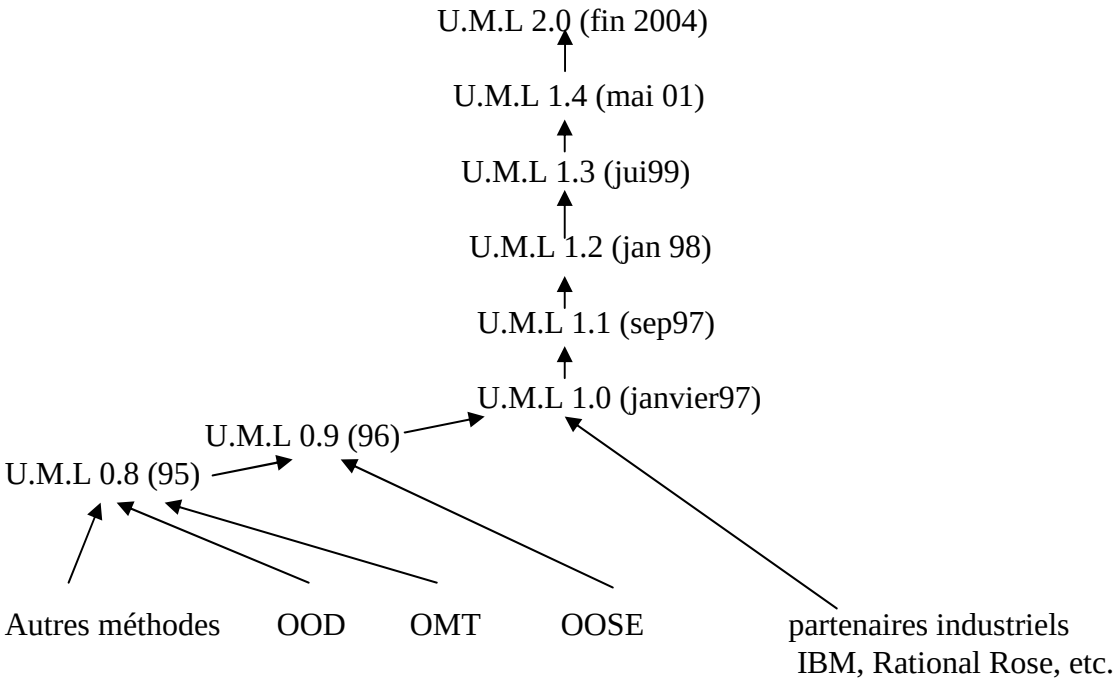
Il existe plusieurs méthodes

- Les méthodes fonctionnelles (1ère génération) : SADT, SA-SD, …Application du principe « diviser pour régner
- Les méthodes systémiques (2ème génération) : MERISE. Séparation données / traitements, niveaux conceptuels, organisationnels, physiques.
- Les méthodes objets (3ème génération) : OMT, OOA, OOD, HOOD, OOSE, …fusion (UML).

Au début des années 90, plusieurs méthodes objets existaient (environ 50) ayant en commun la notion d'objets, de classes etc.
Chacune avait sa propre notation, ses points forts et faibles, son orientation vers un domaine d'application.
L'idée d'une certaine unification des méthodes objets est née. C'était la définition d'un langage commun unique, utilisable par toutes les méthodes, dans toutes les étapes du cycle de vie et compatible avec les techniques de réalisation actuelles.

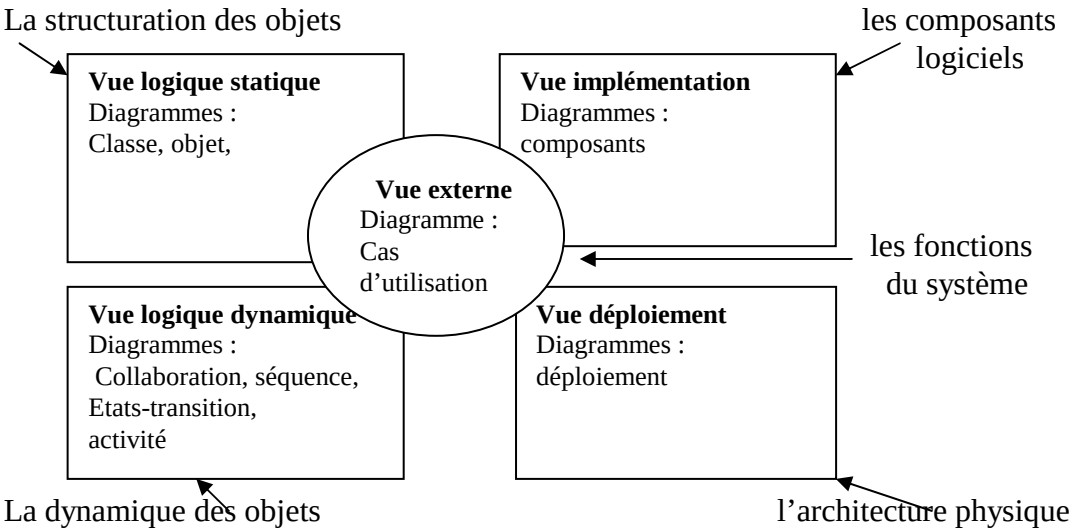
11.2 U.M.L

Ce langage commun s'appelle U.M.L (Unified modeling Language). Il est basée essentiellement sur les méthodes OOD (de Booch), OMT (de Rumbaugh) et OOSE (de jacobson)



U.M.L se propose de créer un langage de modélisation utilisable à la fois par des humains (forme graphique) et des machines (syntaxe précise).
U.M.L n'est pas une méthodologie orientée objet, mais un langage de description des objets.

U.M.L peut être utilisé à toutes les étapes du cycle de vie.



Toutes les vues proposées par U.M.L sont complémentaires les unes des autres, elles permettent de mettre en évidence les différents aspects du logiciel ou le système à réaliser.

11.3 Les cas d'utilisation

- Formalisés par Ivar Jacobson dans OOSE, servent à exprimer le comportement du système en termes d'actions et de réactions, selon le point de vue de chaque utilisateur.
- Spécifient ce que fait le système, mais pas comment il le fait.
- Constituent un moyen pour déterminer les besoins du système.
- Permettent d'impliquer les utilisateurs dès les premières étapes du développement pour exprimer leurs besoins.
- Servent d'entente entre les développeurs, les utilisateurs et les experts sur les fonctions disponibles et sur la façon d'interagir avec le système.

Les fonctions du système?

- Le système devra réaliser X ➔ X est une fonction du système.
- On classe les fonctions en trois catégories: évidentes, cachées, optionnelles.

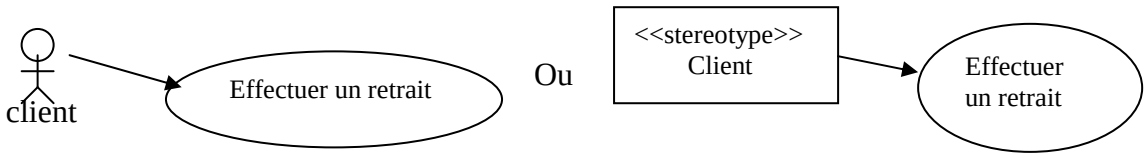
Exemple:
Cas d'un guichet automatique.

Identifier le client : fonction évidente
Mise à jour des soldes des comptes : fonction cachée
Mise à jour du carnet du client : fonction optionnelle

Qu'est ce qu'un cas d'utilisation?
Un cas d'utilisation est un document narratif qui décrit la séquence d'évènements dans laquelle un acteur utilise le système pour accomplir un processus.
Qu'est ce qu'un acteur?
Un acteur est une personne ou un système qui interagit avec le système étudié en échangeant des informations (en entrée et en sortie)

Un acteur représente un rôle joué par un utilisateur qui interagit avec le système. Une même personne physique peut jouer le rôle de plusieurs acteurs et plusieurs personnes peuvent jouer le même rôle.

Exemple



Cas d'utilisation : effectuer un retrait
Acteur : client
Description : un client se présente au guichet, s'identifie, retire un montant et quitte avec son argent.

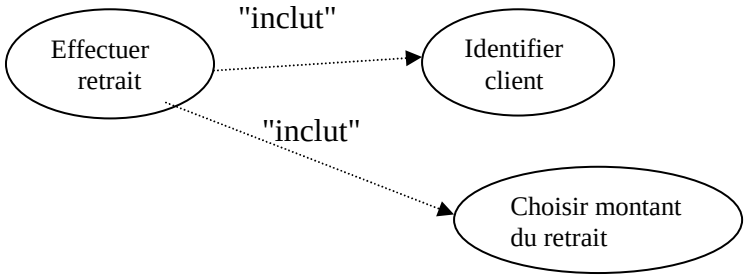
Les cas d'utilisation sont considérés comme des classes de scénarios. Ils peuvent entretenir entre eux les relations suivantes :

- Inclut : un cas inclut obligatoirement un autre
- Etend : un cas et une variante d'un autre.
- Généralisation / spécialisation

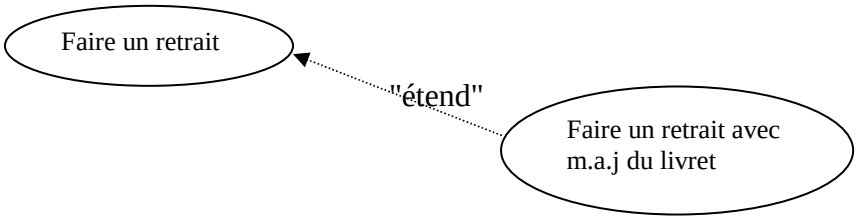
La seule relation qui existe entre les acteurs est la généralisation: un acteur A est une généralisation d'un acteur B si l'acteur A peut être substitué par l'acteur B.

Exemples

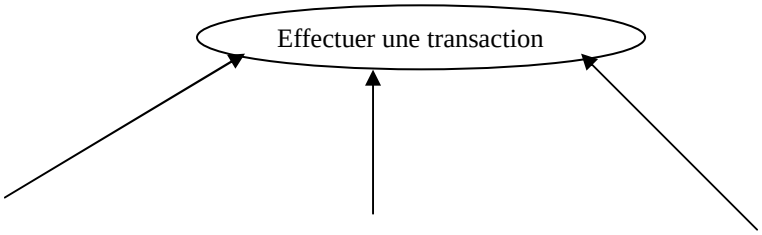
Cas d'inclusion

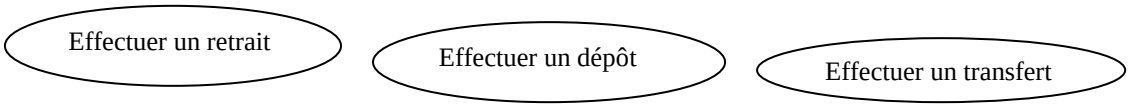


Cas d'extension



Cas de généralisation





11.4 Les interactions

Une application est une société d'objets qui collaborent pour réaliser les fonctions de l'application. Deux types de diagrammes strictement équivalents l'un à l'autre sont utilisés. Ils contiennent la même information, mais représentée de manière différente

11.4.1 Les diagrammes de collaboration

- Ils mettent l'accent sur les relations « spatiales » entre les objets. Les messages peuvent être numérotés pour introduire une dimension temporelle.
- Ils sont souvent utilisés pour décrire la réalisation des cas d'utilisation.

Exemple

L'objet A envoie un message X à un objet B, puis l'objet B envoie un message Y à un objet C, et enfin C s'envoie à lui-même un message Z.

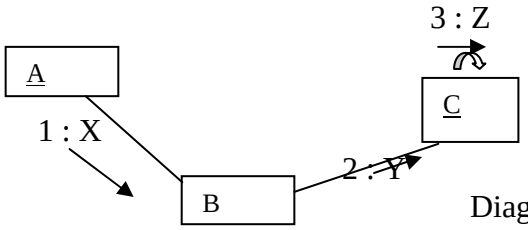
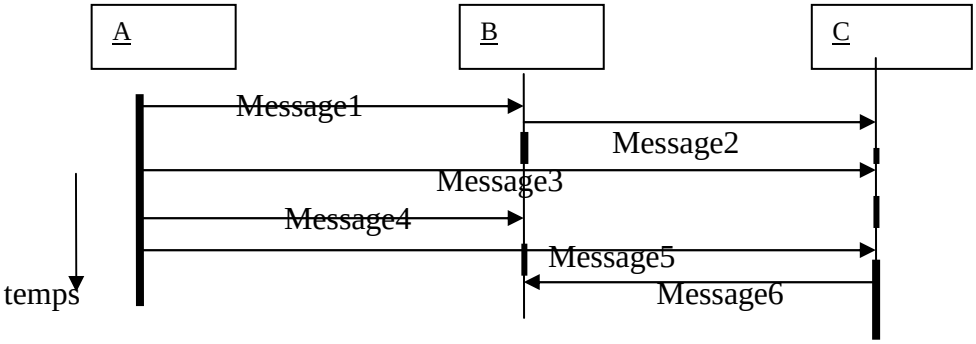


Diagramme de collaboration

11.4.2 Les diagrammes de séquence

- Ils mettent l'accent sur les relations temporelles.
- Ils peuvent être utilisés pour préciser la réalisation des cas d'utilisation ainsi que la dynamique d'un ensemble d'objets.



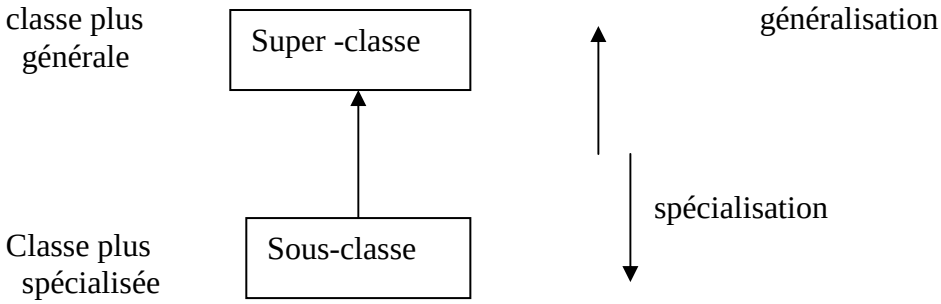
- Chaque réception de message donne lieu à une durée d'activation : le temps de traitement du message.
- La durée d'activation de l'émetteur recouvre celle du récepteur.

11.5 Les classes

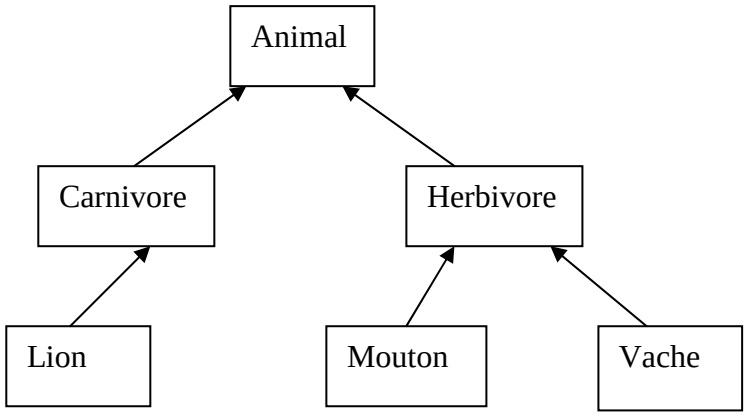
- Regrouper les objets suivant des critères de ressemblance s'appelle classer.
- La classe représente la description abstraite d'un ensemble d'objets. C'est une factorisation des éléments communs à un ensemble d'objets.
- Identification des classes, attributs et des opérations.

- Les diagrammes de séquences et/ou de collaboration permettent d'identifier les objets. Ils sont transformés en classe s'il y a un message sinon en attributs.
- Les opérations des classes correspondent aux messages échangés entre objets réels.

- La hiérarchisation des classes consiste en :
  - la généralisation, dans un sens, qui correspond à la factorisation des éléments communs de classes (attributs, opérations)
  - la spécialisation, dans l'autre sens, qui permet d'adapter une classe générale à un cas particulier.



Exemple



On peut dire :
Tout lion est un carnivore. On parle aussi de relation est-un ou isa en anglais.
Les instances de la classe spécialisée héritent de la description des attributs (variables) et des opérations (méthodes) de la super-classe

- La relation de généralisation /spécialisation est une relation non réflexive, non symétrique et transitive.
- L'héritage multiple est possible.

11.6 Les diagrammes de classes

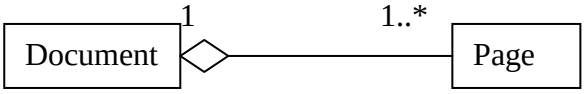
- C'est le plus important dans un développement orienté objet. Il représente l'architecture conceptuelle du système.
- Expriment une structure statique / structurel du système et décrivent l'ensemble des classes et leurs associations

- Une association exprime une connexion sémantique bidirectionnelle entre classes.
- Le rôle décrit une extrémité d'une association.
- Les cardinalités (ou multiplicités) indiquent le nombre d'instances d'une classe pour chaque instance de l'autre classe.

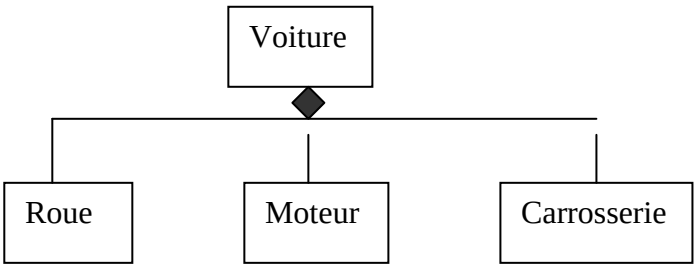
1	un et un seul
0..1	zéro ou un
M..N	de m à n
*	plusieurs
0..*	de zéro à plusieurs
1..*	de un à plusieurs

- L'agrégation :
  - association qui décrit une relation d'inclusion entre une partie et un tout (agrégat).
  - Elle est symbolisée par un **losange plein** pour des objets de durée de vie liée (composition)
  - Elle est symbolisée par un **losange vide** pour des objets de durée de vie indépendante (agrégation).

Exemple



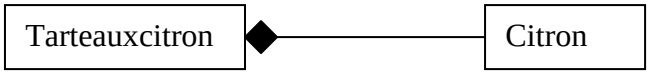
La durée de vie de l'agrégat est indépendante des agrégés.



La durée de vie du composant est dépendante de celle des composés

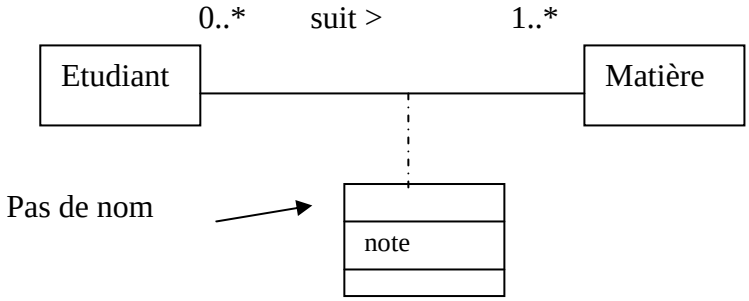
Remarque
Ne pas confondre généralisation/spécialisation et l'agrégation. Quand une classe est une spécialisation d'une autre classe, elle est de même nature, ce qui n'est pas le cas avec l'agrégation.

Exemple

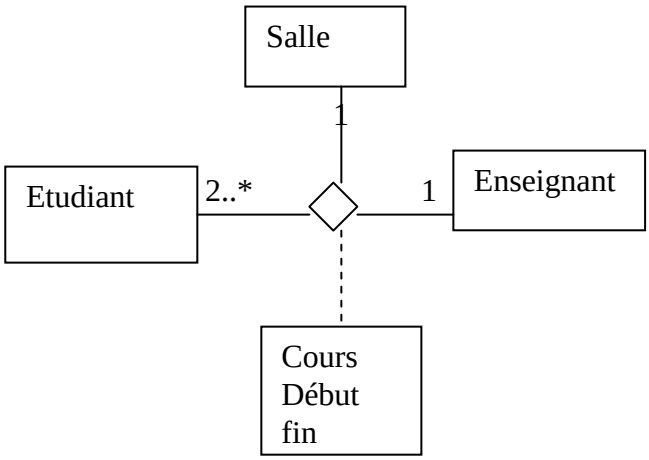


Autres concepts:

- Classe association : association porteuse d'attributs ou d'opérations représentée comme une classe anonyme.



- Association d'arité n : représenté par un losange avec n pattes auquel peut être associé une classe porteuse d'attributs et d'opérations.



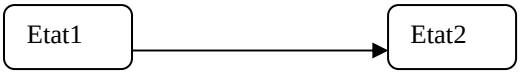
11.7 Les diagrammes d'état transition

- Servent à décrire le comportement des objets (classes) complexes.
- Un état représente un ensemble des valeurs des attributs et des liens avec les autres objets. Il est caractérisé par une durée et il doit être nommé.

- Deux états particuliers existent :
  - état initial représenté par ●
  - Etat final représenté par : ◉

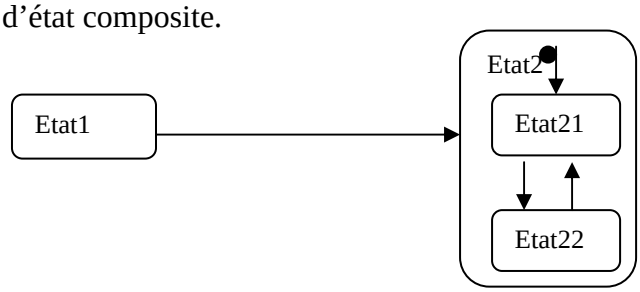
- Une transition représente une connexion entre deux états. Elle assure le passage d'un état à un autre
- Un évènement représente quelque chose qui arrive à un instant connu ou inconnu (interruption, envoi de message). Il peut être chargée d'information. C'est le déclencheur d'une transition.

transition



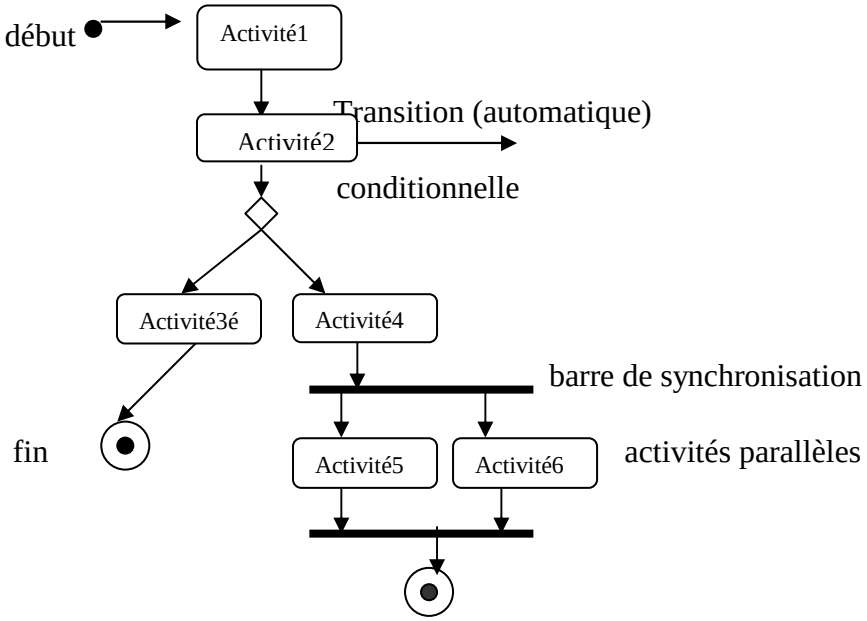
état de départ état d'arrivée

- Ces diagrammes permettent aussi la décomposition d'états en sous états. On parle d'état composite.



11.8 Les diagrammes d'activité

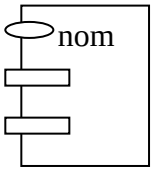
- Représentent un cas particulier des diagrammes état transition dans lequel chaque état correspond à une activité constituant un élément d'une tâche globale à réaliser.
- Permettent de décrire les fonctionnalités et complètent les cas d'utilisation au niveau de l'analyse des besoins.



11.9 Autres diagrammes

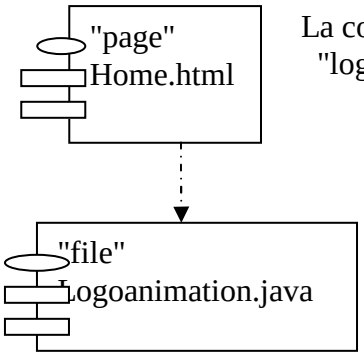
11.9.1 Le diagramme des composants

- Sert à décrire les composantes logicielles et leur interdépendance.
- Les composants sont des codes (source, exécutables), des fichiers, des tables, etc.
- Une composante est représentée par un grand rectangle avec deux petits rectangles placés à gauche du grand rectangle.



- Les dépendances entre les composantes sont illustrées par une ligne pointillée terminée par une flèche simple allant de la composante dépendante vers l'autre composante.
- Une dépendance signifie qu'une composante a besoin d'une autre composante pour que sa définition soit complète.

Exemple

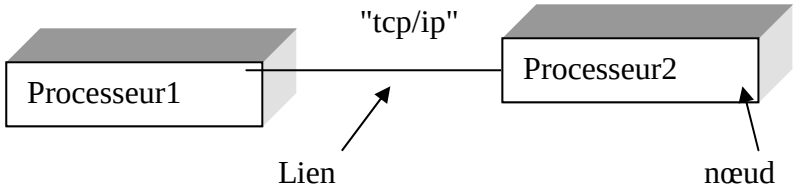


La composante "home.html" dépend de la composante "logoanimation.java".

11.9.2 Le diagramme de déploiement

- Il est la représentation finale de la topologie du système car il unit les facettes hardware et software du système.
- Il doit permettre de visualiser chaque nœud de la topologie du système et d'y voir les composantes logicielles qui s'y exécutent de même que les structure logiques qui sont implantées dans ces composantes.
- Les supports de communication entre les nœuds sont aussi mentionnés.
- Les composantes apparaissant dans ce diagramme doivent obligatoirement figurer dans le diagramme des composantes.
- Les nœuds représentent les unités physiques (matérielles) dotés d'une puissance de calcul (processeurs, imprimantes, périphériques de communication).
- Un nœud est représenté par un cube avec le nom du nœud inscrit à l'intérieur.

Exemple



11.10 Elément d'une démarche

- U.M.L n'impose pas de processus. La démarche naturelle impliquée par la notation U.M.L part des cas d'utilisation qui expriment un point de vue fonctionnel sur le système.

- Puis les diagrammes de collaboration et de séquence associés aux cas d'utilisation font apparaître les classes d'objets impliquées dans le système et donc passer à une vue objets.

Diagrammes essentiels par phase

- Analyse des besoins : cas d'utilisation.
- Analyse et conception du système : diagramme de classes, de collaboration, de séquence, d'activités, d'états, de composants, de déploiement.

11.11 Conclusion

- U.M.L possède une notation simple et facile à apprendre par des non informaticiens.
- La démarche de modélisation adoptée par ce langage est souple, incrémentale et itérative ce qui permet de converger efficacement vers une bonne analyse et une bonne conception.
- La version présentée est la version 1.4 d'UML, la version 2.0 comprend, au total, treize diagrammes

Approche classique	**Approche orientée objet**
Le système est un ensemble processus	Le système est un ensemble d'objets interactifs
Les processus interagissent avec des entités de données.	Les objets interagissent entre eux et avec des personnes.
Les processus reçoivent des entrées et produisent des sorties.	Les objets envoient et reçoivent des messages.

Choix entre une conception classique et une conception orientée objet

Si implémentation orientée objet **alors**
Conception orientée objet *continuité méthodologique*
Sinon
Si application temps réel **alors**
Conception orientée objet * application mieux décrite qu'en classique*
Sinon
Si interface homme machine prépondérante **alors**
Conception orientée objet * existence de bibliothèques*
Sinon
Conception classique
Fsi
Fsi
Fsi

12. Langage de contraintes objet *(Object Constraint Langage : OCL)*

12.1 Introduction

- C'est avec OCL (*Object Constraint Language*) qu'UML formalise l'expression des contraintes. Il s'agit donc d'un langage formel d'expression de contraintes bien adapté aux diagrammes d'UML, et en particulier au diagramme de classes.
- OCL existe depuis la version 1.1 d'UML et est une contribution d'IBM. OCL fait partie intégrante de la norme UML depuis la version 1.3 d'UML. Dans le cadre d'UML 2.0, les spécifications du langage OCL figurent dans un document indépendant de la norme d'UML, décrivant en détail la syntaxe formelle et la façon d'utiliser ce langage.
- OCL peut s'appliquer sur la plupart des diagrammes d'UML et permet de spécifier des contraintes sur l'état d'un objet ou d'un ensemble d'objets comme
  - des invariants sur des classes ;
  - des préconditions et des postconditions à l'exécution d'opérations (les préconditions doivent être vérifiées avant l'exécution, les postconditions doivent être vérifiées après l'exécution);
  - des gardes sur des transitions de diagrammes d'états-transitions ou des messages de diagrammes d'interaction ;
- OCL est un langage typé dont les types sont organisés sous forme de hiérarchie. Cette hiérarchie détermine comment différents types peuvent être combinés. Par exemple, il est impossible de comparer un booléen (*Boolean*) avec un entier (*Integer*) ou une chaîne de caractères (*String*). Par contre, il est possible de comparer un entier (*Integer*) et un réel (*Real*) car le type entier est un *sous-type* du type réel dans la hiérarchie des types OCL.

12.2 Pourquoi OCL ?

Les contraintes peuvent être écrites en langage naturel car il est simple à mettre en œuvre et compréhensible par tous. Par contre il est ambigu et imprécis, il rend difficile l'expression des contraintes complexes.

OCL est un langage formel volontairement simple d'accès. Il possède une grammaire élémentaire (OCL peut être interprété par des outils)

OCL représente, en fait, un juste milieu entre le langage naturel et un langage très technique (langage mathématique, informatique, ...). Il permet ainsi de limiter les ambiguïtés, tout en restant accessible.

12.3 Écriture des contraintes

- Une contrainte constitue une condition ou une restriction sémantique exprimée sous forme d'instruction dans un langage textuel qui peut être naturel ou formel.
- Une contrainte peut être attachée à n'importe quel élément de modèle ou liste d'éléments de modèle
- Une contrainte désigne une restriction qui doit être appliquée par une implémentation correcte du système.

On représente une contrainte sous la forme d'une chaîne de texte placée entre accolades ({}). La chaîne constitue le corps écrit dans un langage de contrainte qui peut être :

- naturel ;
- dédié, comme OCL ;
- ou encore directement issu d'un langage de programmation.

12.4 Représentation des contraintes et contraintes prédéfinies

Soit le diagramme suivant. Il nous servira de support aux différents exemples de contraintes.

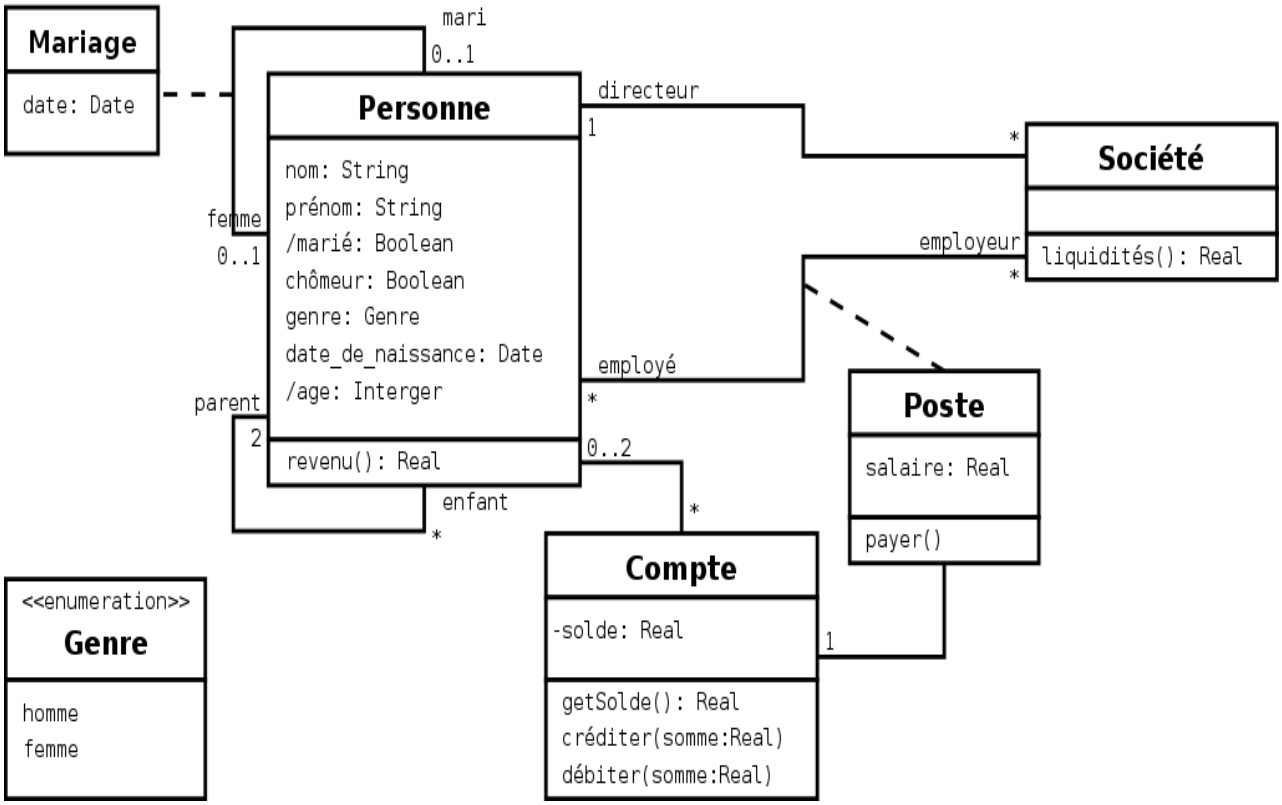


Diagramme de classes modélisant une entreprise et des personnes.

Ce diagramme introduit un nouveau type de classeur, stéréotypé *«enumeration»*, permettant de définir une énumération. Une énumération est un type de donné UML, possédant un nom, et utilisé pour énumérer un ensemble de littéraux correspondant à toutes les valeurs possibles que peut prendre une expression de ce type.

Contexte

Une contrainte est toujours associée à un élément de modèle. C'est cet élément qui constitue le contexte de la contrainte. Il existe deux manières pour spécifier le contexte d'une contrainte OCL :

- En écrivant la contrainte entre accolades ({}) dans une note. L'élément pointé par la note est alors le contexte de la contrainte.

- En utilisant le mot-clef `context` dans un document accompagnant le diagramme.

Syntaxe
`context <élément>`
`<élément>` peut être une classe, une opération, etc.

Exemple
Le contexte est la classe *Compte*:
context Compte

Invariants (*inv*)
Un invariant exprime une contrainte prédicative sur un objet, ou un groupe d'objets, qui doit être respectée en permanence.
Syntaxe
`inv : <expression_logique>`
`<expression_logique>` est une expression logique qui doit toujours être vraie.

Exemple
Le solde d'un compte doit toujours être positif.
context Compte
inv : solde > 0

12.5 Types et opérations utilisables dans les expressions OCL

Types et opérateurs prédéfinis
Le langage OCL possède un certain nombre de types prédéfinis et d'opérations prédéfinies sur ces types. Ces types et ces opérations sont utilisables dans n'importe quelle contrainte et sont indépendants du modèle auquel sont rattachées ces contraintes.

Le tableau ci-dessous donne un aperçu des types et opérations prédéfinis dans les contraintes OCL. Les tableaux suivants rappellent les conventions d'interprétation des opérateurs logiques.

L'opérateur logique *if-then-else-endif* est un peu particulier. Sa syntaxe est la suivante
```
if <expression_logique_0>
then <expression_logique_1>
else <expression_logique_2>
endif
```
Cet opérateur s'interprète de la façon suivante : si *<expression_logique_0>* est vrai, alors la valeur de vérité de l'expression est celle de *<expression_logique_1>*, sinon, c'est celle de *<expression_logique_2>*.

Type	Exemples de valeurs	Opérateurs
Boolean	*true* ; *false*	*and* ; *or* ; *xor* ; *not* ; *implies* ; *if-then-else-endif* ; ...
Integer	1 ; −5 ; 2 ; 34 ; 26524 ; ...	* ; + ; − ; / ; *abs*() ; ...
Real	1,5 ; 3,14 ; ...	* ; + ; − ; / ; *abs*() ; *floor*() ; ...
String	"To be or not to be ..."	*concat*() ; *size*() ; *substring*() ; ...

Types et opérateurs prédéfinis dans les contraintes OCL.

*E*1	*E*2	*P*1 *and P*2	*P*1 *or P*2	*P*1 *xor P*2	*P*1 *implies P*2
vrai	vrai	vrai	vrai	faux	vrai
vrai	faux	faux	vrai	vrai	faux
faux	vrai	faux	vrai	vrai	vrai
faux	faux	faux	faux	faux	vrai

Interprétation des quatre connecteurs, *E*1 et *E*2 sont deux expressions logiques.

expression	*not expression*
vrai	faux
faux	vrai

Convention d'interprétation de la négation.

Collections

OCL définit également la notion d'ensemble sous le terme générique de collection (*collection* en anglais). Il existe plusieurs sous-types du type abstrait *Collection* :

Ensemble (*Set*) :
collection non ordonnée d'éléments uniques (*i.e.* pas d'élément en double).

Ensemble ordonné (*OrderedSet*) :
collection ordonnée d'éléments uniques.

Sac (*Bag*) :
collection non ordonnée d'éléments identifiables (*i.e.* comme un ensemble, mais pouvant comporter des doublons).

Séquence (*Sequence*) :
collection ordonnée d'éléments identifiables.

Jusqu'à UML 2.0 exclu, les collections étaient toujours plates : une collection ne pouvait pas posséder des collections comme éléments. Cette restriction n'existe plus à partir d'UML 2.0.

Accès aux propriétés et aux objets dans les contraintes OCL

Dans une contrainte OCL associée à un objet, il est possible d'accéder aux propriétés (attributs, opérations et terminaison d'association) de cet objet, et donc, d'accéder de manière transitive à tous les objets (et leurs propriétés) avec qui il est en relation.

Accès aux attributs et aux opérations (*self*)
Pour faire référence à un attribut ou une opération de l'objet désigné par le contexte, il suffit d'utiliser le nom de cet élément. L'objet désigné par le contexte est également accessible par l'expression *self*.
On peut donc également utiliser la notation pointée : `self.<propriété>`.
Une opération peut avoir des paramètres, il faut alors les préciser entre les parenthèses de l'opération.
Lorsque la multiplicité d'un attribut, de type *T*, n'est pas 1 (donc s'il s'agit d'un tableau), la référence à cet attribut est du type *ensemble* (*i.e. Set*(*T*)).

Par exemple, dans le contexte de la classe *Compte*, on peut utiliser les expressions suivantes :

- solde
- **self.**solde
- getSolde()
- **self.**getSolde()
- débiter(1000)
- **self.**débiter(1000)

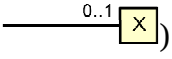
Dans l'exemple précédent, le résultat de l'expression ***self.****débiter(1000)* est un singleton du type *Real*. Mais une opération peut comporter des paramètres définis en sortie ou en entrée/sortie. Dans ce cas, le résultat sera un tuple contenant tous les paramètres définis en sortie ou en entrée/sortie.

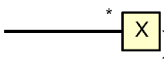
Navigation via une association

Pour faire référence à un objet, ou un groupe d'objets, en association avec l'objet désigné par le contexte, il suffit d'utiliser le nom de la classe associée (en minuscule) ou le nom du rôle d'association du coté de cette classe. Quand c'est possible, il est préférable d'utiliser le nom de rôle de l'association du côté de l'objet auquel on désire faire référence. C'est indispensable s'il existe plusieurs associations entre l'objet désigné par le contexte et l'objet auquel on désire accéder, ou si l'association empruntée est réflexive.

Le type du résultat dépend de la propriété structurelle empruntée pour accéder à l'objet référencé, et plus précisément de la multiplicité du côté de l'objet référencé, et du type

de l'objet référencé proprement dit. Si on appelle *X* la classe de l'objet référencé, dans le cas d'une multiplicité de :

- 1, le type du résultat est *X* (ex : 

) ;
- * ou 0..*n*, ..., le type du résultat est *Set*(*X*) (ex : 

) ;

Emprunter une seule propriété structurelle peut produire un résultat du type *Set*
Emprunter plusieurs propriétés structurelles peut produire un résultat du type *Bag* (ou *Sequence*).
Exemple

dans le contexte de la classe *Société* **(context** Société) :

- `directeur` désigne le directeur de la société (résultat de type *Personne*) ;
- `employé` désigne l'ensemble des employés de la société (résultat de type *Set*(*Personne*)) ;
- `employé.compte` désigne l'ensemble des comptes de tous les employés de la société (résultat de type *Bag*(*Compte*)) ;
- `employé.date_de_naissance` désigne l'ensemble des dates de naissance des employés de la société (résultat de type *Bag*(*Date*)).

Navigation vers une classe association

Pour naviguer vers une classe association, il faut utiliser la notation pointée classique en précisant le nom de la classe association en minuscule.
Par exemple,
Pour accéder au salaire de tous les employés, il faut écrire
context Société:
self.poste.salaire

Cependant, dans le cas où l'association est réflexive (c'est le cas de la classe association *Mariage*), il faut en plus préciser par quelle extrémité il faut emprunter l'association. Pour cela, on précise le nom de rôle de l'une des extrémités de l'association entre crochets ([]) derrière le nom de la classe association.
Par exemple,
pour accéder à la date de mariage de toutes les femmes, il faut écrire :
context Personne
self.mariage[femme].date

Navigation depuis une classe association

Il est tout-à-fait possible de naviguer directement depuis une classe association vers une classe participante.
Exemple :
context Poste
inv : self.employé.age > 21

Par définition même d'une classe association, naviguer depuis une classe association vers une classe participante produit toujours comme résultat un objet unique.
Par exemple, l'expression *self.employé.age* de l'exemple précédant produit bien un singleton.

Opérations sur les collections

Introduction du : «.», «->» et *self*

Pour accéder aux propriétés (attributs, terminaisons d'associations, opérations) d'un objet, OCL utilise la notation pointée : <*objet*>.<*propriété*>.
Cependant, de nombreuses expressions ne produisent pas comme résultat un objet, mais une collection. Le langage OCL propose plusieurs opérations de base sur les collections. Pour accéder ce type d'opération, il faut, utiliser non pas un point mais une flèche : <*collection*> -> <*opération*>.

«.» –
permet d'accéder à une propriété (attributs, terminaisons d'associations, opérations) d'un objet ;

«->» –
permet d'accéder à une propriété d'une collection.

L'objet désigné par le contexte est également accessible par l'expression *self*.
self n'est pas uniquement utilisé pour désigner le contexte d'une contrainte dans une expression mais également pour désigner le contexte d'une sous-expression dans le texte (en langage naturel). Ainsi, lorsque l'on utilise *self* pour une opération <*opération*>, c'est pour désigner l'objet (comme une collection par exemple) sur lequel porte l'opération. Cette objet peut être le résultat d'une opération intermédiaire comme l'évaluation de l'expression <*expression*> précédant l'opération <*opération*> dans l'expression complète : <*expression*>.<*opération*>.

Opérations de base sur les collections
Nous décrivons ici quelques opérations de base sur les collections que propose le langage OCL.

size():Integer –
retourne le nombre d'éléments (la cardinalité) de *self*.

includes(objet:T):Boolean –
vrai si *self* contient l'objet *objet*.

excludes(objet:T):Boolean –
vrai si *self* ne contient pas l'objet *objet*.

includesAll(c:Collection(T)):Boolean –
vrai si *self* contient tous les éléments de la collection *c*.

excludesAll(c:Collection(T)):Boolean –
vrai si *self* ne contient aucun élément de la collection *c*.

isEmpty() –
vrai si *self* est vide.

notEmpty() –
vrai si *self* n'est pas vide.

Opérations de base sur les ensembles (*Set*)

Nous décrivons ici quelques opérations de base sur les ensembles (type *Set*) que propose le langage OCL.

union(set:Set(T)):Set(T) –
retourne l'union de *self* et *set*.

intersection(set:Set(T)):Set(T) –
intersection entre *self* et *set*.

including(objet:T):Set(T) –
Le résultat contient tous les éléments de *self* plus l'objet *objet*.

excluding(objet:T):Set(T) –
Le résultat contient tous les éléments de *self* sans l'objet *objet*.

Exemples

1. Une société a au moins un employé :
   context Société **inv :** self.employé -> notEmpty()

2. Une société possède exactement un directeur :
   context Société **inv :** self.directeur -> size()=1

3. Le directeur est également un employé :

   context Société **inv :** self.employé -> includes(self.directeur)

Opération sur les éléments d'une collection

Syntaxe générale

La syntaxe d'une opération portant sur les éléments d'une collection est la suivante :
<collection> -> <opération> (<expression>)
Dans tous les cas, l'expression <*expression*> est évaluée pour chacun des éléments de la collection <*collection*>. L'expression <*expression*> porte sur les propriétés des éléments en les citant directement par leur nom. Le résultat dépend de l'opération <*opération*>.

Opération **select** et **reject**

Ces deux opérations permettent de générer une sous-collection en filtrant les éléments de la collection *self*. Leur syntaxe est la suivante :

```
select( [ <élément> [ : <Type> ] | ] <expression_logique> )
reject( [ <élément> [ : <Type> ] | ] <expression_logique> )
```

select
permet de générer une sous-collection de *self* ne contenant que des éléments qui satisfont l'expression logique <*expression_logique*>.

reject
permet de générer une sous-collection contenant tous les éléments de *self* excepté ceux qui satisfont l'expression logique <*expression_logique*>.

Par exemple,
pour écrire une contrainte imposant que toute société doit posséder, parmi ses employés, au moins une personne de plus de 50 ans, on peut écrire indifféremment :

1. **context** Société
   inv: self.employé->select(age > 50)->notEmpty()

2. **context** Société
   inv: self.employé->select(individu | individu.age > 50)->notEmpty()

3. **context** Société
   inv: self.employé->select(individu : Personne | individu.age > 50)->notEmpty()

Opération ***forAll* et *exists***

Ces deux opérations permettent de représenter le quantificateur universel et le quantificateur existentiel. Le résultat de ces opérations est donc du type *Boolean*.
Leur syntaxe est la suivante :

```
forAll( [ <élément> [ : <Type> ] | ] <expression_logique> )
exists( [ <élément> [ : <Type> ] | ] <expression_logique> )
```

forAll

permet d'écrire une expression logique vraie si l'expression <*expression_logique*> est vraie pour tous les éléments de *self*.

exists

permet d'écrire une expression logique vraie si l'expression <*expression_logique*> est vraie pour au moins un élément de *self*.

Par exemple
Pour écrire une contrainte imposant que toute société doit posséder, parmi ses employés, au moins une personne de plus de 50 ans, on peut écrire :
context Société
inv: self.employé->exists(age > 50)

L'opération *forAll* possède une variante étendue possédant plus d'un itérateur. Dans ce cas, chacun des itérateurs parcourra l'ensemble de la collection. Concrètement, une opération *forAll* comportant deux itérateurs est équivalente à une opération *forAll* n'en comportant qu'un, mais réalisée sur le produit cartésien de *self* par lui-même.

Par exemple
Pour imposer qu'il n'existe pas deux instances de la classe *Personne* pour lesquelles l'attribut *nom* a la même valeur, c'est à dire pour imposer que deux personnes différentes ont un nom différent, on peut écrire :

1. **context** Personne
   inv: Personne.allInstances()->forAll(p1, p2 | p1 <> p2 implies p1.nom <> p2.nom)

12.6 Conclusion

- Ocl Est un langage non autonome : il est toujours associé au langage UML.
- Ocl permet de préciser davantage les modèles UML.
- Ocl est un outil utilisé dans l'approche dite ingénierie dirigée par les modèles (IDM).

Les langages de programmation

Le but de ce chapitre est de fournir aux étudiants des différents paradigmes de programmation. Le chapitre est organisé comme suit:

1. Introduction
2. L'évolution des langages de programmation
3. Le style de programmation
4. Implantation des langages de programmation
5. Le choix d'un langage de programmation
6. Rôle de l'ingénieur en logiciel
7. Conclusion

1. Introduction

- Les langages de programmation jouent un rôle fondamental en informatique et tout système informatique doit être réalisé avec un ou plusieurs langages.
- La principale caractéristique des nouvelles approches de programmation est la facilité d'expression et de compréhension par les humains.
- Les langages de programmation
  - bien définis syntaxiquement, sémantiquement
  - implémentés efficacement et
  - bien adaptés au domaine d'application

  permettent le développement des logiciels sur des bases raisonnables.
- Il existe un très grand nombre de langages de programmation (>2000)

2. L'évolution des langages de programmation

- Langage machine :
  - est apparu tout à fait au début (< 1950)
  - seul langage compréhensible par la machine.
  - dispose des instructions du type:
    - transfert de données: entre registres, mémoire et périphériques
    - arithmétique: +, -, *, / sur des entiers ou flottants
    - contrôle: poursuivre l'exécution à un autre point du programme conditionnellement ou non

  Avantages:
  - contrôle total sur la machine (accès à toutes les possibilités de la machine).
  - Possibilité d'exécution rapide.

  Inconvénients:
  - Non compréhensible par les humains.
  - Le langage change d'un type de machine à une autre.

- Langage d'assemblage
  - représentation symbolique du langage machine.
  - utilise des mnémoniques pour nommer les instructions (ADD, MOV, ..) et on peut nommer les emplacements mémoire et les constantes.
  - Un peu lisible avec un codage peu efficace (beaucoup de code à écrire pour faire peu de choses.
  - Ce sont des langages de bas niveau où il faut s'occuper davantage des détails ayant plus de rapport avec la machine qu'avec le calcul à faire.

- Langage de haut niveau
  - langage qui facilite la tâche de la programmation en cachant les détails se rapportant à la machine.

  Avantages:
  - Notation familière (facilite l'écriture, la compréhension, la modification du programme)
  - Portabilité (possibilité d'un programme d'être utilisé sur d'autres machines avec peu de changements).
  - Réutilisation de librairies de code.
  - Détection automatique d'erreurs.

  Tout cela contribue à augmenter la productivité des programmeurs.

  D'autre part, un langage qui en 1960 était de haut niveau peut paraître aujourd'hui de bas niveau, donc tout est relatif.

3. Le style de programmation

Le style de programmation représente la façon d'exprimer le calcul.

Exemple

Le calcul de **n!**

Style impératif

```
Int  I, f;
f = 1;
 I = 2;
While (I <= N)
  {
     f = f * I;
     I = I + 1
   }
```

Style fonctionnel

```
Int f( Int  N)
 {
   if (N = 0)
     return 1
   else
    return  N * f(N-1);
    }
```

3.1 La programmation imperative

- Le calcul est une séquence d'actions. L'action de base est l'affectation (modification de la valeur associée à une variable du programme.
- C'est le type de langage le plus utilisé.
- Les langages basés sur ce principe sont nombreux tels que l'Assembleur, le Fortran, l'Algol, le Pascal.
- La majorité des langages de programmation utilisés en génie logiciel sont de ce type:
  - existence du savoir-faire
  - existence des outils de commandes
  - efficacité d'exécution
  - introduction à la modularité.
- Ce sont des langages compilés.

3.2. La programmation applicative et la programmation fonctionnelle

Dans la programmation applicative,

- l'action de base est le calcul d'une expression plus précisément le calcul du résultat d'une fonction pour des valeurs de ses arguments.
- le premier langage est Lisp dont sont issues plusieurs variantes.
- Ce sont des langages interprétés.
- Ce sont des langages très utilisés en I.A.

Dans la programmation fonctionnelle,

- les variables peuvent être de type fonction donc on peut effectuer des calculs sur ces variables comme pour les autres types.
- Ils sont peu utilisés en génie logiciel car ils ne sont pas très efficaces et n'offrent pas de véritables modularités.
- Ils sont utilisés à des fins d'enseignement et de recherche et en milieu industriel Et à des fins de maquettages.

3.3 La programmation logique

- Elle est basée sur le principe de résolution; La résolution signifie la recherche des variables qui rendent l'expression vraie (les solutions).
- Le langage le plus utilisé est Prolog (Programmation logique); Il existe plusieurs extensions et variantes.
- Ces langages présentent d'intéressantes perspectives en génie logiciel pour le maquettage et la production des jeux de tests.

3.4. La programmation à objets

- La programmation est basée sur la définition des classes permettant à l'exécution de créer des instances (objets) qui interagissent par des appels à des méthodes.
- Le premier langage est Simula, destiné à la simulation et au maquettage.
- Ce type de langages et ses variantes sont surtout utilisés en I.A.

3.5 Les langages spécialisés

Mathématiques, simulation : Matlab
Web : Javascript
Description : HTML, XML
Bases de donnés : Cobol, SQL
Etc….

4. Implantation des langages de programmation

4.1 Langages compilés

Exemple : **Pascal**

Avantages:

- rapidité d'exécution
- code source inutile à l'exécution
- détection des erreurs à la compilation

Inconvénients :

- recompilation à chaque modification

4.2 Langages interprétés

Exemple : **Basic**

Avantages

- test, développement

Inconvénients:

- rapidité
- présence d'interpréteur pendant l'exécution.
- interprétation ligne par ligne.

4.3 Langages compilés et interprétés

Exemple : **Java, Perl**

Avantages

- Code intermédiaire portable
- Plus rapide que les langages interprétés

Inconvénients

- Vitesse d'exécution
- Présence d'interpréteur

5. Choix d'un langage de programmation

Il peut dépendre des contraintes du client, des contraintes du projet, du niveau d'abstraction, des habitudes et des bibliothèques existantes. Mais, certains facteurs peuvent influencés sur le choix du langage pour une certaine tâche tels que:

- simplicité conceptuelle : plus le langage est simple, plus il est facile de le maîtriser.
- Puissance d'expression : le langage supporte t-il plusieurs styles de programmation?
- Support de gros projets : le langage offre t-il des mécanismes pour aider la réalisation des gros projets.
- Performance : compilateur rapide ? exécutable rapide?
- Portabilité : le langage est t-il répandu?
- Fiabilité : le langage aide t-il à détecter les erreurs. Détection des erreurs à la compilation ? à l'exécution?
- Coût de développement et de maintenance : les programmeurs disponibles maîtrisent t-ils le langage? Sont ils productifs? Le langage permet t-il de réutiliser les librairies et programmes existants?

Exemple:

Pourquoi le choix d'un langage orienté objet (C++, Java, etc.)?

L'orienté objet favorise le développement

- de logiciels qui peuvent être combinés
- de logiciels réutilisables
- de logiciels extensibles

Ce sont **des qualités souhaitées** pour un logiciel.

6. Rôle de l'ingénieur en logiciel

- La programmation à petite échelle (**in-the-small**) :
  Structures de données, **algorithmes,** langages de programmation.
- La programmation à grande échelle (**in-the-large**):
  - traduction du **cahier de charges** vagues en spécifications précises
  - capable de parler à l'utilisateur en termes d'application et non de problème d'ordinateur.
  - capacité de modéliser et d'utiliser de modèle comme référence.
  - capacité d'organisation du projet et de planification.

7. Conclusion

- Ecrire des programmes clairs, compréhensibles et bien documentés.
- Choisir des langages de programmation permettant d'aboutir à des logiciels de qualité (extensibles, réutilisables, compatibles, etc.).
- Utiliser plus qu'un langage s'il est nécessaire en choisissant le langage approprié pour chaque partie de l'application.

Test et preuve d'un programme

Le but de ce chapitre est de fournir aux étudiants des différentes techniques de tests et la preuve d'un programme. Le chapitre est organisé comme suit:

1. **Introduction**
2. **Définition**
3. **Caractérisation du test**
4. **Le coût du test**
5. **Classification des tests**
6. **Les techniques de tests**
7. **Les tests d'intégration**
8. **Les techniques formelles**
9. **Conclusion**

1. Introduction

- But du test : arriver à **un produit 0 défaut.**
- Généralement 40% du budget global est consacré à l'effort de test.
- D'après la terminologie de l'IEEE (norme 729), la **faute** est à l'origine de l'**erreur** qui se manifeste par des **anomalies** dans le logiciel qui peuvent causer des **pannes.**

Fautes ➔ erreur ➔ anomalies ➔ panne

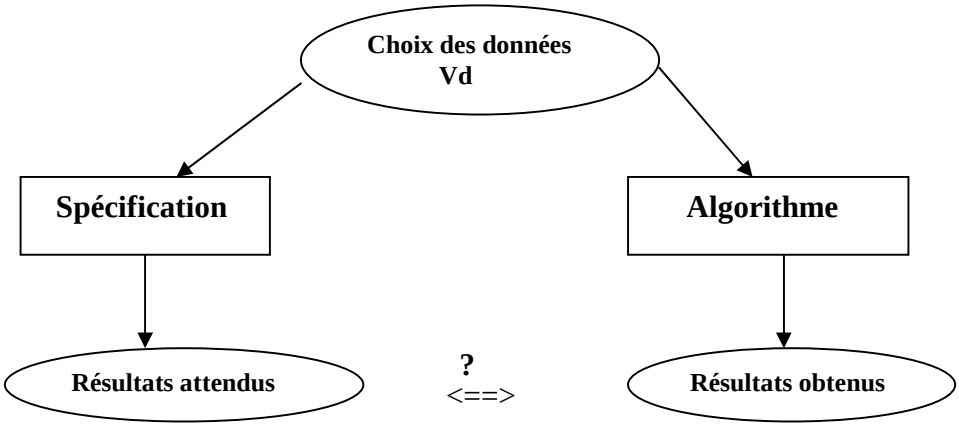
La faute est la cause de l'erreur.
L'erreur : écart entre une valeur ou une condition calculée, observée ou mesurée et la valeur ou la condition qui est vraie, spécifiée ou théoriquement correcte.
L'anomalie ou défaut: la manifestation de l'erreur dans le logiciel. Un défaut peut causer une panne.
La panne: la fin des capacités d'un système ou d'un de ses composants d'effectuer la fonction requise ou de l'effectuer à l'intérieur des limites spécifiées.

Pour garantir la qualité, il est nécessaire d'effectuer les activités de validation et de vérification.
La validation a pour fonction la mise en évidence des défauts.
Valider c'est répondre à la question "faisons nous le bon produit?"
La vérification a pour fonction la mise en évidence des erreurs.
Vérifier c'est répondre à la question "faisons nous le produit correctement?"

Quel impact?
Sécurité des personnes, retours des produits, relations contractuelles, etc.

2. Définition (norme IEEE729)

Le test est un processus **manuel ou automatique** qui vise à établir
qu'un système vérifie les propriétés exigées par sa spécification
ou à détecter des différences entre les résultats engendrés par le système et ceux qui sont attendus par la spécification.



- Le test vise à mettre en évidence les erreurs d'un logiciel.
- Le test n'a pas pour objectif de
  - déterminer les causes des erreurs,
  - corriger les fautes,
  - prouver la correction d'un programme.

Tester, **c'est exécuter** le programme dans l'intention d'y **trouver des anomalies ou des défauts.**

Exemple :

L'algorithme A décrit ci-dessus calcule la racine carrée entière d'un nombre n par recherche dichotomique

```
Données : n : entier ; résultat : r : entier
Var rmin, rmax, m : entier
Début
Rmin := 0 ;
Rmax := n + 1 ;
Tantque  rmin + 1 < >  rmax  faire
  m := (rmin + rmax)  div  2
   Si m * m  >  n  alors
     Rmax := m
   sinon
     Rmin := m
  Finsi
Fintantque
 R := rmin.
```

L'application d'un test (Vd, Vr) sur A consiste

- à exécuter A en lui fournissant la valeur Vd et
- à vérifier que la valeur V fournie par l'exécution est identique à la valeur attendue Vr
- Si V = Vr ➔ le test est réussi, sinon le test a échoué et A contient une erreur.

Les couples (0, 0), (1, 1), (4, 2), (6, 2) sont des tests réussis, alors que (5, 3) est un test qui échouerait.

2.1 Qu'est ce qu'une spécification ?

On appelle spécification d'un algorithme A un quadruplet formé de :

- une désignation et un typage de données.
- une désignation et un typage des résultats.
- une pré condition définissant des contraintes sur les données restreignant les valeurs admises par le typage.
- une post condition définissant le résultat en fonction des données.

Exemple

La spécification pour l'algorithme A est la suivante :

Données n : entier

Résultat r : entier

Pré condition : $n > 0$

Post condition $(r^2 \leq n) \Lambda (n < (r + 1)^2)$

3. Caractérisation du test

Un test comprend :

- une étape de génération de jeux de test : un programme est une application d'un domaine d'entrée E vers un domaine de sortie S. La génération de jeux de test sélectionne un sous domaine de E.
- une étape d'exécution : fourniture des entrés au programme, récupération des sorties pour l'analyse.
- Une étape d'observation des résultats (oracle) : observation par un humain ou automatique.

Le but est de répondre à la question : l'exécution a t-elle donné de bons résultats ?

4. Le coût du test

Selon [Pressman 92], la répartition du budget global est la suivante :

- Analyse et conception : 40%
- Codage : 20%
- Test et installation : 40%

5. Classification des tests

C'est une classification selon le cycle de vie

- Test unitaire : test d'un programme ou d'un module dans le but de s'assurer qu'il ne comporte pas d'erreurs d'analyse ou de programmation.
- Test d'intégration : Tous les modules assemblés sont testés (incluant le code importé).

- Test système : teste le système comme un seul bloc. Le test système inspecte les fonctionnalités du logiciel.
- Test de réception : test généralement effectué par l'acquéreur dans ses locaux après installation d'un système ou une unité fonctionnelle avec la participation du fournisseur pour vérifier que les dispositions contractuelles sont respectées.
- Test de non regression : à la suite de la modification d'un logiciel (ou un de ses constituants), un test de non regression a pour but de montrer que les autres parties du logiciel n'ont pas été affectées par cette modification.
- Test de robustesse : il consiste à tester le système dans des conditions extrêmes, grandes quantités de données ou grand nombre d'usagers.

6. Les techniques de tests

on retrouve les tests statiques, les tests dynamiques et les tests aléatoires.

6.1 Les tests statiques

Les méthodes de tests statiques consistent en l'analyse textuelle du code du logiciel afin de détecter des erreurs sans exécution du programme.

6.1.1 Revue de code

Revue ou inspection (IEEE 729)
Examen détaillé d'une spécification ou d'une conception ou d'une implémentation par une personne ou un groupe de personnes afin de déceler des défauts, des violations de normes de développement ou d'autres problèmes.
Elle s'effectue en trois étapes:

- préparation : recherche des défauts
- cycle de réunions
- suivi : vérification des corrections ou nouvelle inspection

6.1.2 Walkthrough

L'équipe chargée de la revue du code va, en plus de la revue, simuler son exécution sur un jeu de test construit et décrit dans des documents standardisés.

6.1.3 La recherche d'anomalies

Exemple
Typage impropre, incohérence entre les interfaces des modules, déclaration des arguments jamais utilisés, une expression ayant une valeur clairement constante.
Les outils utilisés sont les suivants:

- Le graphe de contrôle
- Le flot ou lot de données
- L'évaluation symbolique

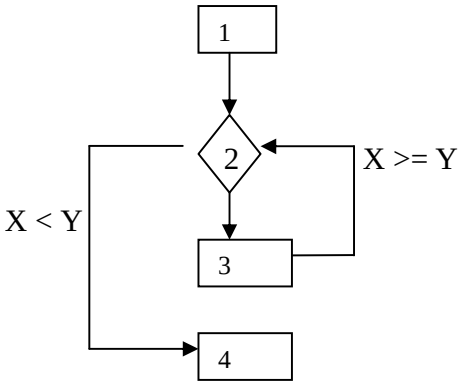
a) Le graphe de contrôle

- Identifier les blocs d'instructions indivisibles maximaux.
- Chaque bloc n'a qu'un seul point d'entrée : son début
- Les sommets du graphe de contrôle sont les blocs déjà définis et les conditions qui apparaissent dans les instructions conditionnelles ou les boucles
- Les arcs qui les relient correspondent aux transferts de contrôles possibles entre ces sommets.
- Si un bloc est constitué par "if- then – else" ➔ 3 sommets et 2 arcs
- Si présence d'une boucle ➔ un circuit.

Exemple

```
1 Open fichier1;
  Read ( X, Fichier1);
  Read (Y, fichier1);
  Z: = 0;
2 While X >= Y do
3 Begin
   X: = X - Y;
   Z: = Z + 1;
  End
4  Print (Y, Fichier2);
  Close Fichier1;
  Close Fichier2;
```

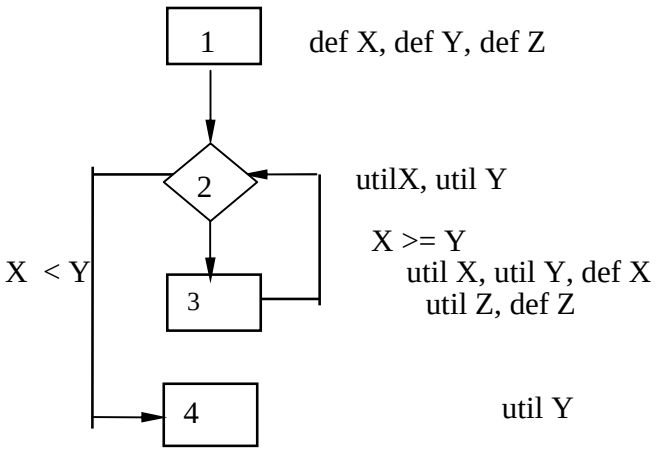
Le graphe de contrôle est le suivant:



On considère tous les chemins du point d'entrée au point de sortie et on cherche les anomalies. Par exemple, présence de code isolé, enchaînement logique de certaines instructions (écrire sur un fichier fermé, ne pas fermer un fichier).

b) Flots de données

Annoter le graphe par des informations pertinentes quant à la manipulation des variables (définition d'une variable, utilisation d'une variable).



Les principales anomalies sont les suivantes :

- Utiliser une variable sans la définir au préalable.
- Définir une variable sans l'utiliser ultérieurement.

Exemple

Le long chemin est 1.2.3., la variable Z est définie en 1 mais jamais utilisée. Même chose pour le chemin 1.2.3.2.4

La difficulté liée à l'usage de chemins extraits du graphe de contrôle ou du flots de données est l'existence de chemins infaisables.

Exemple

```
Read (X);
V:= 2*X;
If X >0 then
  Y:= X + 1
   Else
   Z:= 2*X;
If V < 0 then
 T:= Y
  Else
  T:=Z;
```

L'anomalie détectée: le chemin passant par le 1er then (Y:=X + 1) et le 2ème else
c-a-d T:= Z
La variable Z est utilisée mais non définie: c'est un chemin infaisable

Pour les systèmes complexes, il est extrêmement difficile de savoir si un chemin est faisable ou non c'est un problème indécidable ➔ pas de solution générale.

c) L'évaluation symbolique
Construction des expressions symboliques reliant les données d'entrée à celles de sorties ainsi que les conditions pour suivre un certain chemin lors de l'exécution.

Exemple

```
Read (X) ;
Y := X +1 ;
If Y <= 0 then
  Z:= Y * X
  else
  Z := Y ;
```

Comme expression symbolique associée à Z

```
(X0 + 1)* X0  si  (X0 +1)  <= 0
  X0 +1       si  (X0 + 1)  > 0
```

Deux problèmes peuvent être soulevés :
Problème 1 posé : taille des expressions manipulées ➔ simplification➔ facilite la mise en _oeuvre de ces méthodes.
Problème 2 posé : programme avec une infinité de chemins : boucles ➔ résultats avec un nombre donné d'itérations sinon utilisation d'un invariant de boucle (cas de preuve de programme).

6 .1.4 Avantages et inconvénients des méthodes de tests statiques
Avantages
- Méthodes efficaces et peu coûteuses
- 60 à 95% des erreurs sont détectées lors des contrôles statiques

➔ Les méthodes de tests statiques sont nécessaires

Inconvénients
- Méthodes ne permettant pas de valider le comportement d'un programme au cours de son exécution.

- Lors de la modification d'un programme, il est difficile de réutiliser les tests précédents pour valider la nouvelle version

➔ Les méthodes de tests statiques sont insuffisantes

6.2 Les méthodes de tests dynamiques

- Ils reposent sur l'exécution du logiciel.
- La démarche est la suivante :
  - sélection d'un jeu de test. **Comment concevoir les jeux de test?**
  - exécution du jeu de test.
  - décider du succès ou de l'échec du jeu de test. **Comment décider du succès ou de l'échec du jeu de test ?**

Comment concevoir les jeux de tests ?

6.2.1 Test structurel ou boite blanche

- La sélection des jeux de tests repose sur la description de la structure du logiciel à tester
- La description utilise le graphe de contrôle et le flot de données.
- Tous les chemins reliant le point d'entrée au point de sortie sont considérés.
- La sélection de données d'entrée permet de déclencher l'exécution de certains de ces chemins.

Pourquoi se baser sur **la structure du programme** pour le tester ?

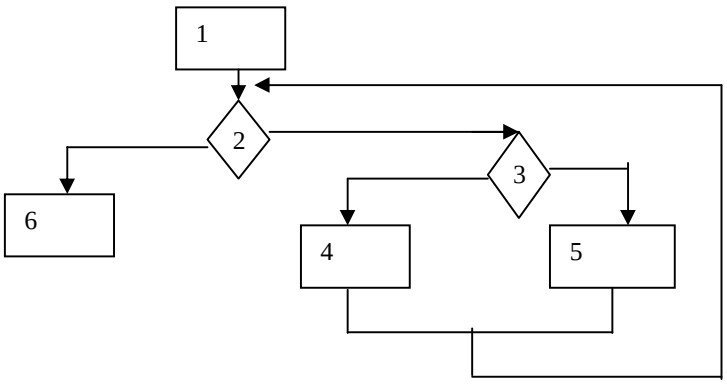
- Les erreurs ont tendance à se concentrer dans des chemins, des instructions, des conditions qui sont hors de l'exécution normale.
- Un chemin particulier a peu de chance d'être exécuté alors qu'il l'est très souvent.

Les critères de sélection sont :

a) Couverture de toutes les instructions où chaque bloc séquentiel d'instructions doit être atteint par au moins un chemin parmi les chemins parcourus.

Exemple

```
1  Read(x);
   Read(y);
2  While (not(x=y)) do
3  If x > y then
4  X:=x – y
   Else
5  Y:=y –x;
6  Pgdc :=x ;
```



Le jeu de test est le suivant :

X=3 ; y=3

X=4 ; y=3

X=3 ; y=4

b) Couverture de tous les arcs

Exemple

Dans le cas de l'exemple précédent, le jeu de test choisi précédemment parcourt tous les arcs.

Remarque

Le critère de couverture de toutes les instructions ne correspond pas au critère de couverture de tous les arcs

Exemple

```
If x <0 then
  X:=-x;
  Z:= x;
```

Le jeu de test x = -2 couvre toutes les instructions **mais** pas tous les arcs.

c) Couverture des chemins du graphe de contrôle

Exemple

```
If (not (x=0)) then
Y:= 5
Else z := z  - x;
If (z > 1) then
Z := z / x
Else
Z:=0;
```

Le jeu de tests couvrant tous les chemins est le suivant:

{(x = 0 ; z = 1), (x = 1 ; z = 3), (x = 0; z = 3), (x = 1; z = 1)}

Il est possible de déterminer le nombre de chemins indépendants à partir du graphe de contrôle.

Un chemin indépendant est un chemin qui parcourt au moins un nouvel arc par rapport aux chemins déjà définis.

La complexité cyclomatique ou nombre cyclomatique (Mc Cabe 76) :

C = nbre- arcs – nbre- nœuds + 2

Ceci représente la borne supérieure du nombre de tests à effectuer pour que tous les arcs soient couverts au moins une fois.

Pour Mc Cabe, la limite de la valeur de la complexité cyclomatique est de 10, mais dans la pratique, la limite supérieure est de 30 environ. Au-delà, la procédure est difficile à tester.

Remarque

Le nombre de chemins peut devenir très grand dans des programmes réels. Ce critère est inexploitable en pratique. Par exemple, dans le cas de boucles (ou boucles imbriquées).

On se contente de couvertures incomplètes comme la couverture **des i-chemins**.

d) Couverture des conditions

Le jeu de test doit couvrir à vrai et à faux toutes les conditions élémentaires de toutes les conditionnelles.

Exemple

If (x > 1 and b = 0) then
X := x / n;
If (a = 2 or x > 1) then
X := x + 1.

Comme jeu de test, on peut choisir
{a = 1, b = 0, x = 3} ; {a = 2, b =1, x =1}
Il couvre toutes les conditions **mais** pas toutes les instructions

Remarque

- Il est nécessaire de mélanger différents critères des tests.
- Ce type de test structurel ne peut pas être réutilisé tel quel en cas de modifications de code.

6.2.2 Test fonctionnel ou boite noire

- sélection du jeu de test à partir d'une spécification de ce que le logiciel doit faire et non pas à partir des textes de programme.
- Ce type de test est moins développé que pour le test structurel; les spécifications sont écrites en langage naturel (textes) ➔ difficile de mettre en évidence des méthodes systématiques de sélection de jeux de tests.
- Au lieu de tester toutes les valeurs, on partitionne les valeurs en classes d'équivalence et on teste aux limites des classes.

Comment partitionner un domaine en classes d'équivalence ?

Exemple1

Soit un programme P, supposons que les données de P soient des nombres appartenant à D = [0 ..5000]

- Une classe d'équivalence pour des valeurs appartenant à l'intervalle des valeurs valides. D1 = 0 <= d <=5000
- Une classes d'équivalence pour des valeurs invalides inférieures
  D2 = d < 0
- Une classe d'équivalence pour des valeurs invalides supérieures
  D3 = d > 5000

Un jeu de test valide : T = {10, -1, 5001}
On a donc à tester les nombres aux limites des classes

Exemple2

```
If  x > y  then
  Max := x
Else
  Max := x;    /* erreur max := y */
```

La partition est la suivante :

D1 = {(x, y) / x > y}

D2 = {(x, y) / x <= y}

Un jeu de test : {(2, 1), (1, 3)}, {6,2), (4, 7)}

Ce critère de sélection permet de trouver des jeux de tests qui vérifient les deux branches de la condition ➔ possibilité de détecter l'erreur.

Mais, l'erreur peut ne pas être détectée.

Contre exemple : {(4, 2), (2, 2)}, {(6, 3), (3, 3)}

6.2.3 Les tests aléatoires

- Le jeu de test est sélectionné au hasard sur le domaine des entrées du programme.
- Le domaine des entrées du programme est déterminé à l'aide de l'interface du programme ou de l'interface de la spécification.

Inconvénients des tests aléatoires

Ces méthodes ne garantissent pas une bonne couverture des entrées du programme: en particulier, elles peuvent ne pas prendre en compte certains cas limites ou exceptionnelles.

Comment décider du succès ou de l'échec d'un jeu de test?

Une fois un jeu de test sélectionné, il est utilisé lors de l'exécution du programme à valider. Il reste à interpréter les résultats obtenus au cours de cette exécution.

C'est **le rôle de l'oracle** qui décide du succès ou de l'échec du jeu de tests.

L'oracle représente **une référence** (par exemple une spécification ou un prototype) qui permettra de prendre cette décision

- Succès d'un jeu de test (jeu de test >0) si chaque test du jeu est >0
- Echec du jeu de test si chaque test du jeu est <0

Pour chaque test élémentaire f et pour un programme P, l'oracle O donne une des 3 réponses suivantes :

f >0 ➔ P satisfait f

f <0 ➔ P ne satisfait pas f

f indécidable ➔ P ne termine pas par l'exemple

Quand estime t-on que le programme n'a pas besoin d'être testé?

Un programme n'a pas besoin d'être testé

- lorsque l'efficacité du jeu de tests sélectionné est conforme à certains critères de qualité et
- lorsque le jeu de test est >0

L'efficacité d'un jeu de test peut être évalué à l'aide de la méthode de mutations de programme.

Cette méthode consiste à générer des programmes incorrects (mutants) par perturbation syntaxiquement corrects du code.
Exemple de défaut

- Changement de constantes, d'indices.
- Modification du sens de décisions (< en <=)
- Suppression / échange d'instructions.

Un défaut donne lieu à un mutant
Exécution du mutant et comparaison avec le programme d'origine :
si résultats différents ➔ le mutant est tué
si même résultat ➔ le mutant survit
Utilité de la technique des mutant : le **meilleur jeu de test** est celui qui tue **le plus de mutants.**

7. Les tests d'intégration

La phase d'intégration intervient après la vérification unitaire des modules. Le plan d'intégration est rédigé pendant la phase de conception globale (cycle de vie en V). Il prévoit comment les modules, une fois vérifiés, sont intégrés les uns aux autres.
Différentes stratégies d'intégration sont possibles dont :

7.1 Intégration globale (bing-bang)

- adaptée aux petites applications
- modules peu dépendants

7.2 Intégration "au fur et à mesure"

- Les modules sont intégrés dès qu'ils sont disponibles
- Impossible de connaître à l'avance les bouchons et les lanceurs.
  Les bouchons : modules simulant les modules appelés depuis ceux en cours d'intégration et
  les lanceurs : modules simulant les modules appelant ceux en cours d'intégration

7.3 Intégration progressive

- incrémentale: intégration des modules selon un ordre pré spécifié, chaque étape ajoute un incrément au résultat de la précédente.

7.3.1 Types d'intégration progressive possibles :

a) Intégration ascendante
   - les composants situés au plus bas sont intégrés en premier
   - disponibilité immédiate des entrées /sorties

b) Intégration descendante
   - les composants situés au plus haut sont intégrés en premier
   - Le squelette du programme validé tôt.

c) Intégration guidée par les risques
   intégrer d'abord les modules ou sous-systèmes présentant les risques élevés en termes de coûts, de ressources humaines, etc.

8. les techniques formelles

- elles permettent de prouver formellement la correction d'un programme
- Un programme donné est caractérisé par sa pré condition et sa post condition.
- Une pré condition : condition éventuellement à respecter par les données d'un programme.
- Une post condition : condition vraie à la fin du programme qui définit son objectif.
- Pour réaliser la preuve d'un programme, **la logique de Hoare** définit des assertions logiques intermédiaires.

Preuve d'un programme

La preuve constitue un complément aux tests de modules. Elle se pratique sur les modules dont la criticité est suffisamment importante pour que l'on veille à assurer la correction absolue.

La preuve d'un programme repose sur la notion de système formel.

Un système formel est un triplet < L, Ax, R > où

- L : est un langage définissant un ensemble de formules.
- Ax : chaque formule de Ax est un axiome (Ax est un sous ensemble de L)
- R : un ensemble de règles de déduction de formules à partir d'autres formules. Une règle permettant de déduire la formule f à partir de f1, f2, .. fn est notée ainsi : $\frac{f1, f2\ f3, \quad fn}{f}$

8.2.1 Correction partielle d'un programme : la logique de Hoare

La logique de Hoare constitue un système formel. C'est un triplet < L, Ax, R) avec :

- L est l'ensemble des formules {p} A {q}
- Ax est l'ensemble des axiomes de la forme suivante {p(x :=e)} x :=e {p}
- R est l'ensemble des règles de déduction suivant :

{p(x :=e)} x :=e {p} **(AFF)**

~~P' =>P, {P} A {Q}~~ renforcement de la pré condition **(RPre)**
{P'} A {Q}

~~{P} A {Q}, Q => Q'~~ affaiblissement de la post condition **(APost)**
{P} A {Q'}

~~{P} A1 {R}, {R} A2 {Q}~~ **(SEQ)**
{P} A1, A2 {Q}

~~{e & P} A {Q}, p & not e~~ **(SI)**
{P} Si e alors A fsi {Q}

{e & P} A1 {Q}, {not e & P} A2 {Q} **(SINON)**
{P} Si e alors A1 sinon A2 fsi {Q}

{e & I} A {Q} **(TANTQUE)**
{I} Tantque e faire A fintque {I & not e}

En plus des premières règles, Il y a autant de règles que structures de composition des instructions dans le langage algorithmique.

- La règle **(SEQ)** définit la sémantique de la composition séquentielle de deux instructions A1 et A2.
- Les règles **(SI)** et **(SINON)** définissent la sémantique de la conditionnelle dans es deux formes possibles **avec** et **sans** sinon.
- La règle **(TANTQUE)** définit la sémantique de l'itération.

La règle du Tantque est remarquable puisqu'elle met en évidence le fait que chaque pas d'itération maintient l'invariant I.

{I} ← invariant
Tantque e faire
{I & e}
A
{I}
Fintque
{I & not(e)}

8.2.1.1. Propriétés de la logique de HOARE

Elle recouvre deux propriétés :

- la correction : si on démontre le théorème {p} A {q}, alors l'exécution de l'algorithme A à partir de n'importe quelles données qui satisfait le prédicat p aboutit aux résultats qui satisfont le prédicat q.
- La complétude : si l'algorithme A aboutit à des résultats qui satisfont le prédicat q à partir de n'importe quelles données qui satisfont le prédicat p, alors il existe une preuve du théorème {p} A {q}.

Exemple1

{x=1} x:=x + 1; {x=2} T1
{x=1} x:=y; y:=x + 1; {x=y} T2
➔ T1 est valide, T2 ne l'est pas

{x=1} x:=x+1; {x=2} à prouver?
On applique **(AFF)** ➔ {x+1=2} x:=x+1; {x=2}
Si x=1 alors x+1=2 ; On applique **RPre** {x=1} x:=x+1; {x=2}

Exemple2

{x=x0& y=y0} z:=x; x:=y; y:=z; {x=y0 & y=x0} T3

Utilisation de la règle **(SEQ)**
{x=y0 & z=x0} y:=z {x=y0 & y=x0} T3.2
{x=x0 & y=y0} z:=x; x:=y; {x=y0 & z=x0} T3.1

T3.2 est valide par construction
T3.1 sera décomposé en
{y=y0 & z=x0} x:=y { x=y0 & z= x0} T3.11
{x=x0 & y=y0} z:=x {y=y0 & z=x0} T3.12
T3.11 est valide par construction
T3.12 et par application de la règle **(AFF)** qui donne
{y = y0 & x = x0} z := x {y = y0 & z =x0} T.3.11
et ensuite la règle **(RPre)** : la pré condition de T3 implique celle de T.3.11
x = x0 & y = y0 ➔ y = y0 & x = x0

Exemple3
{true} ⟵ pré condition
if x >= y then
max := x
else
max := y.
{max = x or max = y} and {max >= x and max >= y} ⟵ post condition

si x >= y on exécute max : = x
Pour que (max = x or max = y) and (max >= x and max >= y) soit vraie après max := x
Il faut que (x = x or x = y) and (x >= x and x >= y) soit vrai avant c-a-d x >= y ce qui est vrai d'après la condition.
si X < Y on exécute max : = Y
Pour que (max = x or max = y) and (max >= x and max >= y) soit vraie après max := y
Il faut que (y = x or y = y) and (y >= x and y >= y) soit vrai avant.
Or not cond c-a-d x < y ➔ x <= y. C'est donc vrai

Exemple4
{ x >= 0} pré condition
while x > 0 then
x := x –1;
End ;
{x = 0} post condition

Comment trouver l'invariant ?
Il faut trouver un invariant qui avec la négation de la condition (Inv & not cond) donne la post condition. Pour cela, x >= 0 peut être considéré comme invariant
car : x >= 0 and not cond x >0
c-a-d x >=0 and x <= 0 donne bien la post condition x = 0.

Remarque

L'axiome et les règles données ci dessous correspondent à un langage très simple dans le sens où :

- les expressions sont sans effet de bord c-a-d leur évaluation ne modifie pas les valeurs des variables du programme (pas d'appel d'une fonction).
- les variables sont simples (pas d'éléments de tableau ni de pointeurs).

8.2.2 Les preuves de terminaison

- Possibilité de généraliser les logiques de Hoare de manière à vérifier la correction totale du programme.
- Correction totale d'un programme = correction partielle + la preuve de Terminaison.
- Le problème de la terminaison est difficile. Pour démontrer la terminaison d'un programme:
  - Identifier la variable de contrôle de la boucle.
  - Démontrer qu'elle évolue de manière monotone vers la borne (supérieure ou inférieure) qui correspond à la condition d'arrêt.

Enfin, d'une manière générale, elle relève de l'intuition.

9. Conclusion

- Le test logiciel peut être utilisé pour montrer la présence des erreurs, mais jamais pour prouver leur absence.
- La mise en évidence d'une méthode optimale de vérification de programme passe par une combinaison de l'utilisation de différentes méthodes de tests statiques et dynamiques.
- Les tests structurels et les tests fonctionnels sont utilisés de façon complémentaires.
- Les preuves, sous différentes formes, sont surtout pratiquées pour le développement de logiciels critiques.

La maintenance d'un logiciel

Le but de ce chapitre est de fournir aux étudiants des notions sur la maintenance, l'évolution et de la ré-ingénierie. Le chapitre est organisé comme suit:

1. Définitions
2. Les types de maintenance
3. Le processus de maintenance
4. L'estimation du coût de maintenance
5. Les lois de l'évolution des programmes
6. La ré-ingégnierie
7. La documentation
8. Conclusion

1. Définitions

Qu'est ce que la maintenance?

La maintenance est l'ensemble des activités effectuées pour modifier un logiciel après sa mise en opération.

Justifications : la maintenance du logiciel est nécessaire pour

- correction d'erreurs (erreurs résiduelles)
- adaptation aux besoins des utilisateurs,
- amélioration du logiciel (code, conception),
- modernisation du logiciel (remplacement d'un système de gestion de fichiers par une base de données)
- changements d'environnement technique (passage à Microsoft .Net)

Qu'est ce que la maintenabilité?

La maintenabilité est la facilité avec laquelle

- un logiciel peut être corrigé en cas d'erreurs,
- il peut être agrandi, modifié pour satisfaire de nouveaux besoins.

2. Les types de maintenance

La maintenance regroupe des activités différentes qui sont regroupées en 4 catégories:

La maintenance perfective

Elle consiste à améliorer le fonctionnement du logiciel sans changer les missions spécifiées du logiciel.

La maintenance adaptative

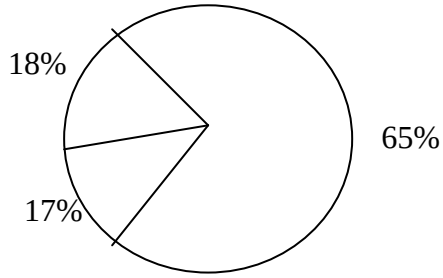
Activité de maintenance qui consiste à modifier un logiciel afin de l'adapter aux changements dans les spécifications des données ou aux évolutions de l'environnement.

La maintenance corrective

Activité de maintenance qui consiste à corriger les défauts résiduels des logiciels (erreurs de programmation, défauts de spécification,….).

La maintenance préventive : consiste à effectuer une "pré maintenance" dans le but de prévenir certains problèmes.

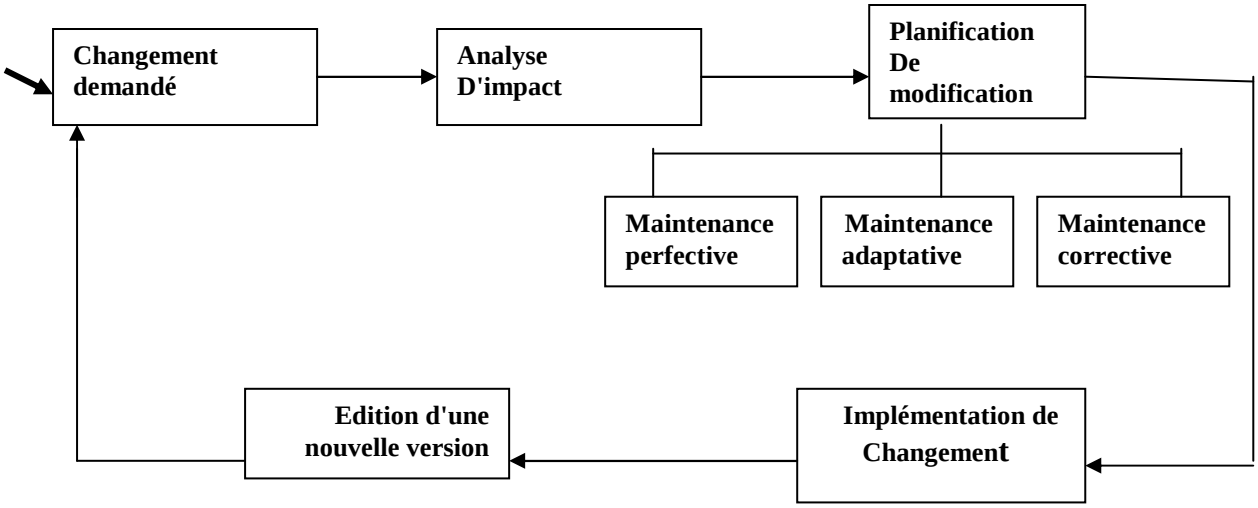
Distribution de l'effort



Maintenance corrective : 17%
Maintenance adaptative : 18%
Maintenance perfective : 65%

3. Le processus de maintenance

Le processus de maintenance peut être schématisé de la façon suivante :



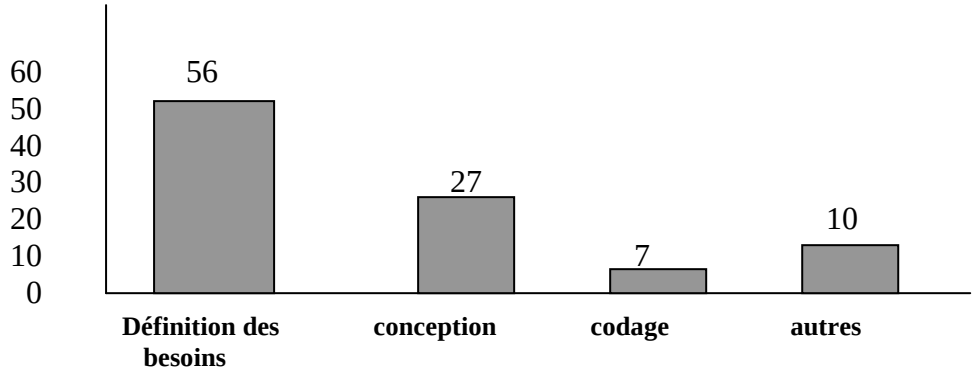
3.1 Informations nécessaires pour la maintenance

a) de l'équipe de développement
   - si elle y est encore
   - analyse : spécifications fonctionnelles
   - listing de sources
   - algorithmes et références
   - dossier de test

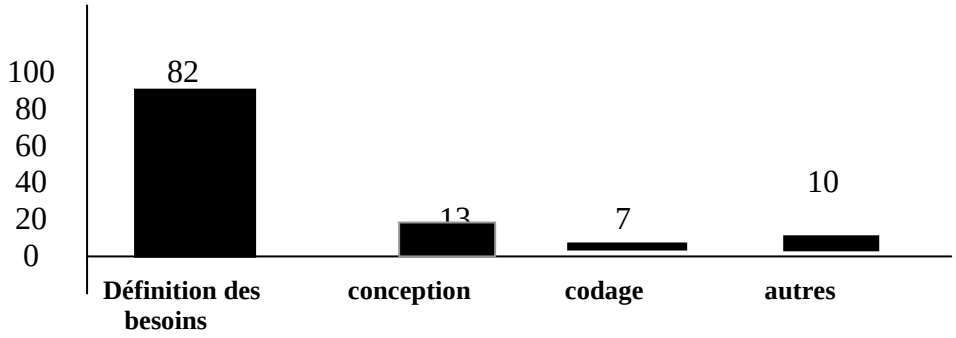
b) de l'utilisateur
   - anomalie ou erreur ou nouvelle fonctionnalité
   - contexte de l'erreur
   - données du client

4.. L'estimation du coût de la maintenance

La majorité des erreurs en génie logiciel ont pour origine la phase "définitions des besoins" du cycle de vie d'un logiciel.



- La répartition du coût de maintenance en génie logiciel



- Donc, 82% du coût total de la maintenance revient aux erreurs dues à des mauvaises spécifications des besoins des utilisateurs.
- Plus les erreurs de spécifications sont découvertes tardivement dans le processus de développement et plus le coût de maintenance est élevé.
- Près de la moitié des erreurs ne sont détectées qu'après la phase des tests.
- Les coûts de maintenance sont habituellement plus grands que les coûts de développement.
- Ils augmentent avec les maintenances successives (la maintenance affecte la structure du logiciel, ce qui rend les maintenances suivantes plus difficiles

Justifications du coût élevé de l'activité de maintenance:

- La stabilité de l'équipe: l'équipe de développement est affectée à de nouveaux projets. Par conséquent, l'équipe de maintenance nécessite beaucoup d'efforts pour comprendre le système et revenir sur les décisions de conception.
- La responsabilité contractuelle: contrat de développement n'inclut pas la maintenance.
- La compétence de l'équipe de maintenance: l'équipe est moins expérimentée et moins familière au domaine d'application.
- La structure et l'âge du programme: le programme est vieux et sa structure tend à se dégrader par les changements ➔ plus dur à comprendre et à maintenir.

5. Les lois de l'évolution des programmes

Les lois de Lehman semblent être applicables aux grands logiciels des grandes organisations. Il n'est pas clair comment les modifier pour les petites et moyennes entreprises ou les systèmes qui utilisent des composantes provenant d'un autre fournisseur.

Les lois de Lehman (cinq lois parmi les huits):

- La loi du changement continuel : un programme utilisé dans un environnement du monde réel doit nécessairement changer sinon il deviendra progressivement inutile dans cet environnement.

- La loi de la complexité croissante : lorsqu'un programme change, sa structure tend à devenir complexe ➔ nécessité de consacrer des ressources additionnelles de façon à maintenir et à préserver sa structure.

- La loi de l'évolution des grands programmes : l'évolution des grands programmes est un processus auto régulateur. Les attributs (taille, temps) entre versions sont approximativement constants pour chaque version.

- La loi de la stabilité organisationnelle : pendant la vie d'un programme, son taux de développement est approximativement constant et indépendant des ressources consacrées au développement du système. En d'autres termes, ça ne sert plus d'augmenter le personnel ou le budget car ça ne progressera pas plus rapidement

- La loi de la conservation de la familiarité : pendant la vie d'un programme, l'incrément de changement dans chaque version est approximativement constant. Donc, pour une période de temps donnée, l'augmentation de la taille et des fonctionnalités du logiciel est limitée et prévisible.

6. La ré-ingénierie

La ré-ingénierie consiste à construire une nouvelle implantation du logiciel à partir de la version existante.

Pourquoi la ré ingénierie? Continuer à maintenir les systèmes hérités: faire du neuf avec du vieux ou en d'autres termes les rendre plus maintenables.

Qu'est ce qu'un système hérité (legacy systems)?

- Système ancien mais fonctionnant toujours et rendant des services importants.
- De très grands changements ont été effectués sur ces systèmes.
- Différentes personnes ont travaillés sur ces systèmes.
- Ce sont des systèmes ayant des spécifications incomplètes, une documentation incomplète ou même absente, écrits avec des langages de programmation anciens et conçus pour matériels anciens.

Face au dilemme suivant:
Faut-il continuer à utiliser le système existant et effectuer des changements nécessaires d'où l'augmentation inévitable des coûts?
Ou bien

Faut-il remplacer le système existant par un nouveau ce qui nécessite un budget important et le nouveau système n'est pas aussi meilleur que l'ancien?

D'où les avantages de la ré-ingénierie:

- La maintenance d'un système existant est plus économique que le développement d'un nouveau logiciel.
- Réduction des risques ➔ fort risque lié au développement d'un système complètement nouveau (spécifications inadéquates, nouveau personnel)
- Réduction des coûts : coût de la ré-ingénierie est souvent plus faible que le coût de développement d'un nouveau logiciel.

6.1 Catégorie de ré-ingénierie

1. Ré-ingénierie du code : le cas le plus fréquent en ré-ingénierie.
   - Structuration du code :
     pas de modification de l'architecture du système
     pas de modification des fonctionnalités des modules
     restructuration du code et/ou des données de chaque module ➔ plus facile à maintenir.
   - Translation du code : conversion d'un langage de programmation à un autre sans autre modification (langage périmé)

2. Ré-ingénierie des données :
   - migration des fichiers de données vers une base de données
   - modèle de données parfois incompréhensible
   - extension des représentations (ex: le bug de l'an 2000 pour modifier la représentation des dates

3. Ré-ingénierie des fonctions :
   - Réorganisation du logiciel : regroupement en un même module des fonctions reliées à un même concept (ex: entées/sorties).

4. Ré-ingénierie du niveau architectural :
   - Passage à un nouveau paradigme (exemple: orientée objet, client / serveur, services Web).
   - Conception périmée ou mauvaise.
   - Réduction du couplage (dépendance) entre les modules.

7. Documentation

Un document du logiciel correspond à n'importe quel artefact destiné à communiquer l'information sur le logiciel.

7.1 L'ensemble des documents se résume en :

- un cahier de charges : besoins
- un dossier d'analyse fonctionnelle
- un manuel de conception
- un dossier de programmation
- un manuel d'installation et de mise en œuvre
- un dossier de procédures de tests

7.2 Qualité des documents

- écriture
- présentation
- complétude
- actualisation

➔ Importante charge

7.3 Quelques conseils

- construire des phrases simples
- faire attention à l'orthographe
- une seule idée par phrase
- paragraphes courts
- termes précis avec glossaire

La documentation doit nécessairement accompagner le logiciel

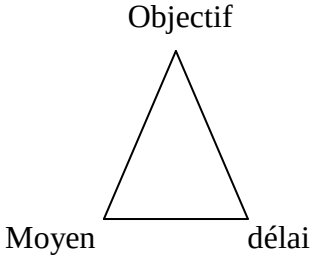
8. Conclusion

- La maintenance ➔ après la livraison du logiciel; Il existe différents types de demandes de maintenance.
- La maintenabilité permet de déterminer l'état de dégradation du logiciel (sa qualité).
- La ré-ingénierie doit être envisagé pour les logiciels utiles mais de qualité faible.
- Le niveau de ré-ingénierie dépend des besoins de l'entreprise et des caractéristiques du logiciel.
- La documentation est souvent négligée bien qu'elle représente un composant très nécessaire du cycle de vie du logiciel.

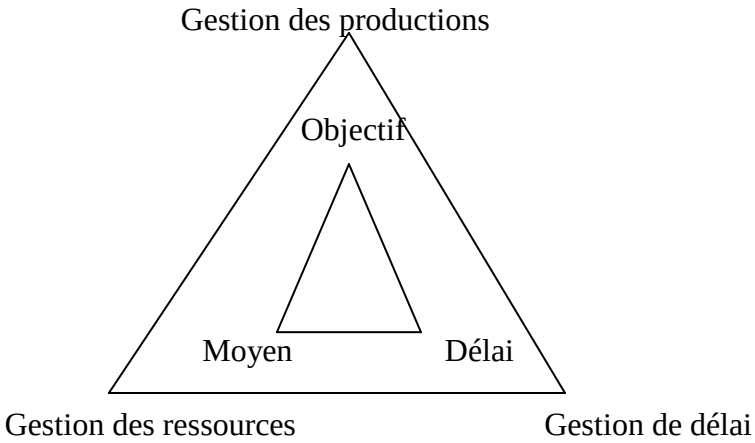
Complément 4 **Gestion de projet**

1. définitions

Qu'est ce qu'un projet ?
C'est un objectif qu'on cherche à atteindre dans un futur proche.
Du point de vue scientifique : c'est l'image d'un futur qu'on espère atteindre.
Du point de vue génie logiciel : c'est le triangle contraint



Qu'est ce que la gestion d'un projet ?
Elle a pour but de le mener à son terme en tenant compte des contraintes qui lient chacun des aspects du triangle projet.



1.1 Types de gestion

- Gestion de délai : elle consiste à déterminer un parcours qu'on va suivre, un calendrier de réalisation
- Gestion de ressources : le moyen constitué du budget du projet est à transformer en travail, matériels, déplacements, locaux, etc.
- Gestion des productions : l'objectif d'un projet consiste à procéder à une ou plusieurs livraisons durant le délai. Ce qui est produit doit se rapprocher du but final.

2. l'estimation de la charge

2.1 Définition
La charge est la quantité de travail qu'une personne peut réaliser. L'unité est : hommes-mois / hommes-jours / hommes-années.
La taille d'un projet se mesure à sa charge, selon la norme ISO

Charge < 6h-m	très petit projet
6 h-m <= charge <= 12h-m	petit projet
12 h-m <= charge <= 30h-m	moyen projet

30 h-m <= charge <= 100h-m grand projet
100 h-m <= charge très grand projet

Exemple :
Charge de 60 h-m ➔ 1 personne en 60 mois (5 ans)
10 personnes en 6 mois
60 personnes en 1 mois

2.2. L'estimation de la charge
Le modèle COCOMO (COnstruct COst Model) de B. Boehm est le lus connu. Il fournit des équations estimant l'effort (ou la charge) en h-m en fonction de la nature du projet.
Ces équations ont été obtenues de façon empirique à partir des statistiques sur des projets réels. Elles sont de la forme :

Charge = a (taille)^b **Délai = c (charge) ^{d}**

Où taille est le nombre d'instructions livrées en milliers de lignes sources.

La taille moyenne d'équipe = charge / délai.
Les paramètres a, b, c, d dépendent de la catégorie du projet.

Type de projet	charge (hommes-mois)	délai (en mois)
Simple	A = 2	C = 2.5
	B = 1.05	D = 0.38
Moyen	A = 3	C = 2.5
	B = 1.12	D = 0.35
Complexe	A = 3.6	C = 2.5
	B = 1.2	D = 0.32

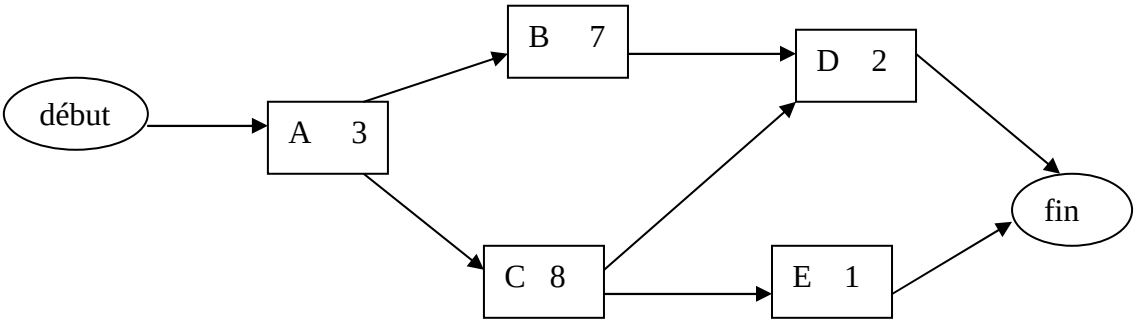
Ces résultats bruts peuvent être affinés en fonction de 14 paramètres tels que l'expérience du langage de programmation, fiabilité, temps d'exécution, etc.
➔ La charge nette = produit (facteurs) * charge brute

3. Planification d'un projet

3.1 Définition
La planification consiste à
- établir la liste des tâches et leur durée,
- déterminer les relations de dépendance entre les tâches,
- déterminer les tâches critiques,
- ordonnancer ces tâches dans le temps.

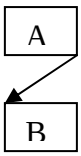
Deux principales techniques sont à la disposition du chef de projet :

3.2 Réseau de PERT (Profit Evaluation and Review Technique)
C'est un graphe acyclique (orienté et sans cycle) qui permet de représenter l'enchaînement des tâches. Chaque nœud du graphe est un couple (Ti, di).
Exemple

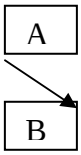


3.2.1 Les types de liens
Il existe 4 types de liens pour l'enchaînement des tâches :

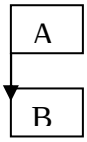
- fin – début : dès que la tâche A est finie, la tâche B commence.



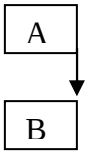
- début - fin : la tâche B ne peut se terminer que si la tâche A commence.



- début – début : La tâche B doit commencer en même temps que la tâche A



- fin – fin : la tâche B doit se finir en même temps que la tâche A.



3.2.2 Paramètres clés

Définition

Pour déterminer le temps de fin de projet, on utilise des paramètres clés (associés à chaque tâche) qui sont des dates au plus tôt (D-tôt, F-tôt) et les dates au plus tard (D-tard, F-tard) ainsi que la marge qui découle logiquement.

Calcul des paramètres

Ce calcul est valable pour les liens de types fin- début.

- Dates au plus tôt :

Si la tâche Ti est au début du projet (t0) alors

D-tôt(Ti) = t0

F-tôt (Ti) = D-tôt(Ti) + di

Sinon

D-tôt(Ti) = max (F-tôt (prédécesseur(Ti))

F-tôt (Ti) = D-tôt(Ti) + di

- Dates au plus tard

Si la tâche Ti est en fin de projet (tf) alors

F-tard(Ti) = tf

D-tard (Ti) = F-tard(Ti) - di

Sinon

F-tard(Ti) = min (D-tard (successeur(Ti))

D-tard (Ti) = F-tard(Ti) - di

Marge : elle s'obtient en faisant la différence entre le temps au plus tard et le temps au plus tôt.

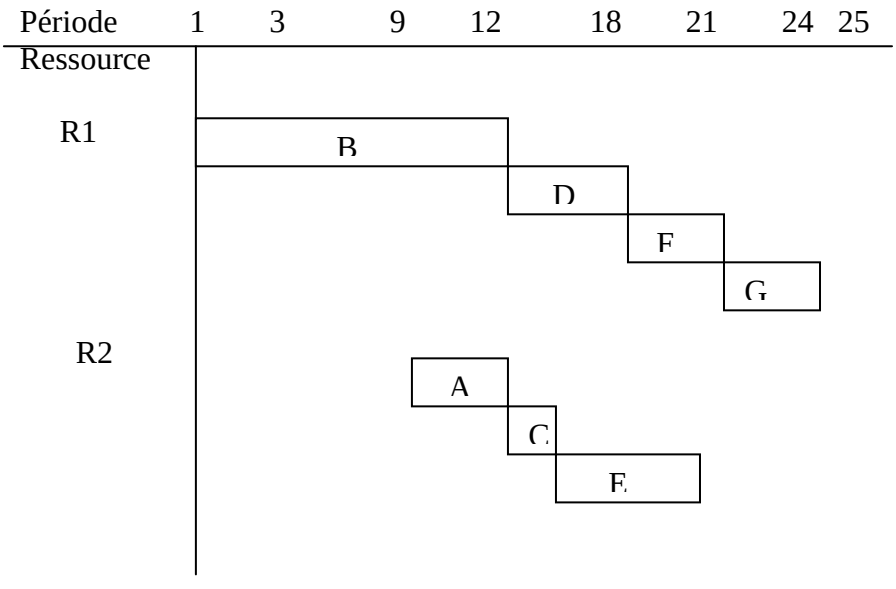
D - tard – D - tôt ; F- tard – F- tôt

Chemin critique

C'est le chemin du graphe ayant les plus petites marges (ou nulle au minimum).

3.3 Diagramme de Gantt

A partir des résultats obtenus du réseau de PERT et des résultats des ressources disponibles, on construit un planning (calendrier) sous forme de diagramme.



4. pilotage du projet

Suivre un projet : c'est un ensemble d'actions à faire :

- mesurer l'état d'avancement
- mesurer ce qui est consommé
- comparer les écarts entre les réalisations et les prévisionnels
- expliquer les écarts
- proposer les actions correctives

➔ suivi individuel et un suivi du projet

4.1 Suivi individuel

Il s'agit de détecter les difficultés éventuelles d'un intervenant à partir du rapport hebdomadaire qu'il doit fournir.

Nom	Tâche	Temps estimé	Temps passé	Reste à faire
*******	Module-1 Réunion	10	3 1	6
******	Programmation Maladie Congé	8	4 1 2	5
*****	Module-2	6	4	2

Comme il est possible d'établir une récapitulation mensuelle de la manière suivante :
Temps passé : T
Reste à faire : R
Avancement : A = calculé ainsi R(n-1) – R(n)

Mois	Tâche	Semaine-1	Semaine-2		Total du mois
		T R A	T R A		
*******	A	4 8 2	4 3 5		

4.2 Suivi du projet

On regroupe les informations globales sur le projet qui servent de base à un point d'avancement périodique. L'intérêt porte sur les tâches même celles qui n'ont pas encore commencé.

Tâche	Mois(n-1)	Mois(n)	Evolution de charge	Total temps passé	Evolution globale	avancement
	T R A	T R A				
A						
B						
C						
……						

Complément 1:

La complexité dans la conception des systèmes complexes

La complexité du logiciel découle de:

- la complexité du problème.
- La complexité de gestion du processus de conception du logiciel par une équipe d'ingénieurs.
- La flexibilité offerte par le logiciel
- La difficulté de décrire les systèmes discrets.

Cette difficulté se traduit par des projets qui:

- se terminent en retard,
- dépassent le budget prévu,
- ne respectent pas les exigences du client.

Les caractéristiques des systèmes complexes:

- Ils adoptent une structure hiérarchique.
- Le choix des composants de base de ces systèmes est souvent arbitraire.
- Les liens intra-composants sont plus forts que les liens inter-composants.
- Ils sont composés d'un nombre restreint de sous systèmes arrangés de diverses manières.
- Ils découlent de systèmes plus simples dont ils sont une évolution.

Les systèmes complexes peuvent se décomposés en systèmes plus simples. Il existe deux principales approches:

- La décomposition algorithmique (classique): la façon dont le problème compte être résolu.
- La décomposition en objets: décomposer le problème en ses différents acteurs qui agissent selon des messages qu'ils reçoivent.

L'approche orientée objet est souvent adoptée. Toute méthode de design orientée objet repose sur:

- une notation permettant de décrire les modèles,
- un processus de construction des modèles,
- des outils facilitant la construction et la description des modèles.

La modularité

La modularité : décomposition du logiciel en composants plus discrets (par exemple en modules).
L'abstraction est un mécanisme qui permet de présenter un contexte en exprimant les éléments pertinents et omettant ceux qui ne le sont pas.

Principe d'ouverture/fermeture

Un module est dit :

- ouvert : s'il est encore disponible pour des extensions/modifications;
- fermé : s'il est prêt à être utilisé par d'autres modules ce qui suppose qu'il est stable.

Critères de modularité:
Une méthode de conception est dite modulaire si elle répond aux cinq critères suivants:

- **Décomposition modulaire**: décomposer le problème en sous problèmes plus simples, reliés entre eux par une structure simple et indépendante de façon à pouvoir travailler sur chacun séparément (SADT).
- **Composition modulaire**: possibilité de produire des éléments logiciels qui peuvent être facilement combinés avec d'autres de façon à produire de nouveaux systèmes dans un environnement différent que celui pour lequel ils ont été initialement développer (réutilisation).
- **Compréhensibilité modulaire**: possibilité de comprendre un module sans avoir à connaître les autres ou au pire après examen de quelques uns.
- **Continuité modulaire**: dans l'architecture créée, un changement de spécification du problème ne déclenche de changement que dans un module ou un petit nombre de modules.
- **Protection modulaire** : l'effet d'une condition anormale se produisant dans un module donné sera confiné dans ce module ou au pire ne se propagera qu'à quelques modules voisins.

La réutilisation:
Les bénéfices attendus par la réutilisation sont:

- **Ponctualité**: réalisation plus rapide, arrivée à temps sur le marché, délai de développement est respecté.
- **Effort de maintenance diminué**: réutiliser un composant logiciel est plus meilleur que de consacrer des efforts pour le maintenir (évolutions futures).
- **Fiabilité**: les composants logiciels sont bien testés et biens validés.
- **Efficacité**: utilisation des meilleurs algorithmes et structures de données possibles dans le domaine spécialisé.
- **Investissement**: préservation du savoir faire et des inventions des meilleurs développeurs

Complément2 **La fiabilité du logiciel**

1. Introduction

- La fiabilité correspond à la probabilité d'effectuer une opération sans panne pendant une durée fixée et pour un contexte bien donné.
- Elle donne une mesure du degré de confiance et elle mesure les conséquences d'une faute.

2. Défaut et faute

- Un défaut est du à la présence d'une faute ; Il a une caractéristique purement dynamique.
- Une faute est une caractéristique statique du logiciel qui provoque un défaut à l'exécution.

Exemple :
Pour un logiciel de saisie, la faute est de ne pas vérifier la mauvaise saisie alors que le défaut serait que le logiciel plante suite à la mauvaise saisie.

3. Amélioration de la fiabilité

Pour assurer la fiabilité du logiciel, des tests sont effectués et des morceaux de programmes sont rajoutés.
➔ Le logiciel devient plus lourd et plus lent et donc moins efficace. Et d'une manière générale, la fiabilité est plus importante que l'efficacité étant donné que l'efficacité est devenue de moins en moins nécessaire vu les prix actuels des machines

4. Métrique de la fiabilité

Probabilité d'une panne
C'est la probabilité que le système se comporte de manière non prévu lorsqu'une requête est effectuée.
Exemple :
P=0.001 ➔ sur 1000 requêtes, on a une probabilité d'un défaut.

Taux de panne
C'est la fréquence d'apparition d'une panne.
Exemple :
T= 0.02 ➔ sur 100 unités, on a 2 pannes.

Temps moyen entre deux pannes
C'est la mesure de temps entre deux apparitions de défauts.
Exemple :
Tm =500 ➔ le temps moyen entre deux défauts est de 500 unités de temps

Disponibilité

C'est probabilité que système soit opérationnel. Le temps de réparation est pris en compte.

Exemple :

Dispo = 0.990 ➔ sur 1000 unités de temps, le système est disponible et utilisable pendant 990 unités de temps

5. Classification des défauts

Classe de panne	Description
Transitoire	Ne se produit qu'avec certaines données
Permanente	Se produit avec toutes les données
Réparable	Ne nécessite pas d'intervention humaine
Irréparable	Nécessite une intervention humaine
Non corruptrice	Ne détruit, ni corrompt les données
Corruptrice	Corrompt les données

6. Conclusion

- la fiabilité est une qualité importante du logiciel.
- Elle est obtenue au détriment de l'efficacité.